

版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！

# 大数据之路

# 阿里巴巴大数据实践

阿里巴巴数据技术及产品部 著

## 架构

## 认知



## 阿里巴巴数据技术及产品部

定位于阿里集团数据中台，为阿里生态内外的业务、用户、中小企业提供全链路、全渠道的数据服务。作为阿里大数据战略的核心践行者，致力于“让大数据赋能商业，创造价值”。

经过多年的实践，数据技术及产品部已经构建了从底层的数据采集、数据处理，到挖掘算法、数据应用服务以及数据产品的全链路、标准化的大数据体系。通过这个体系，超过EB级别的海量数据能够高效融合，并以秒级的响应速度，服务并驱动阿里巴巴自身的业务和外部千万用户的发展。

现在，阿里巴巴数据技术及产品部正通过技术和产品上的创新，探索全域数据的价值，将阿里在大数据上沉淀的能力对外分享，为各行各业的发展带来更多可能性。



阿里数据官网



阿里数据  
微信公众号

# 大数据之路

阿里巴巴大数据实践

|| 阿里巴巴数据技术及产品部 著 ||

電子工業出版社·

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

在阿里巴巴集团内,数据人员面临的现实情况是:集团数据存储已经达到EB级别,部分单张表每天的数据记录数高达几千亿条;在2016年“双11购物狂欢节”的24小时中,支付金额达到了1207亿元人民币,支付峰值高达12万笔/秒,下单峰值达17.5万笔/秒,媒体直播大屏处理的总数据量高达百亿级且所有数据都需要做到实时、准确地对外披露……巨大的信息量给数据采集、存储和计算都带来了极大的挑战。

《大数据之路——阿里巴巴大数据实践》就是在此背景下完成的。本书中讲到的阿里巴巴大数据系统架构,就是为了满足不断变化的业务需求,同时实现系统的高度扩展性、灵活性以及数据展现的高性能而设计的。

本书由阿里巴巴数据技术及产品部组织并完成写作,是阿里巴巴分享对大数据的认知,与生态伙伴共创数据智能的重要基石。相信本书中的实践和思考对同行会有很大的启发和借鉴意义。

本书著作权归淘宝(中国)软件有限公司所有,未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

### 图书在版编目(CIP)数据

大数据之路:阿里巴巴大数据实践/阿里巴巴数据技术及产品部著. —北京:电子工业出版社, 2017.7

ISBN 978-7-121-31438-4

I. ①大… II. ①阿… III. ①企业管理—数据管理 IV. ①F272.7

中国版本图书馆CIP数据核字(2017)第094934号

策划编辑:张彦红

责任编辑:葛娜

印刷:三河市双峰印刷装订有限公司

装订:三河市双峰印刷装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编:100036

开本:720×1000 1/16 印张:21 字数:338千字

版次:2017年7月第1版

印次:2017年7月第1次印刷

印数:4000册 定价:79.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式:(010)51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。



# 本书编委会

编委会主任：朋新宇

主    编：王赛、王永伟

编委会成员（排名不分先后）：

罗金鹏、王静、王赛、王永伟、张磊、赵唯行

本书内容撰写人员（排名不分先后）：

总体架构：王俊华、王永伟

日志采集：李涛、殷霞

数据同步：陈永俊、王俊华

离线数据开发：陈永俊、王永伟

实时技术：黄晓锋

数据服务：方建江、郭才祥、江岚、徐锐

数据挖掘：何覃、王鹏、王中要、徐萧萧、应倩岚、  
郑苏杭

大数据领域建模综述：王赛

阿里巴巴数据整合及管理体系：张子良、王永伟

维度设计：王永伟



事实表设计：陈永俊、方彬、王永伟、张子良

元数据：魏海、王永伟

计算管理：陈中强、贾元乔、孔令娟、袁杰

存储和成本管理：潘旻、王伟

数据质量：方彬

数据应用：肖美丽、郑育杰

**特别感谢（排名不分先后）：**

陈同杰、邓中华、丁燕、何露莎、黄荣、金高平、  
刘凡、李炉阳、刘健男、林鸣晖、李启平、邱坤东、  
梅婧婷、孙伟光、苏艳、王政、王子凌、向师富、  
姚滨晖、杨红霞、张启、张伟、阮城锋、谢丹丹、  
商渭清、裴逸钧、罗鸣、王鹏、冯敏敏、曾文秋等  
对本书出版工作的支持！

# 序

大数据是什么？在过去的 5 年里，恐怕没有另外一个词比大数据更高频；也没有另外一个概念如大数据一样，被纷繁解读，著书立说。有趣的是，作为距离大数据最近的公司之一——尽管我们的初心或许和大数据没有直接关系——在关于大数据的理论和概念的争论中，阿里巴巴却鲜有高谈阔论。

因为自知而敬畏，因为敬畏而谦逊。甚至在大数据这个概念出现很久之前，阿里巴巴就不得不直面、认知、探索，并架构和大数据有关的一切。数据作为一个生态级的平台企业最直接的沉淀，亦是最基本的再生产资料。如果没有基于大数据的人工智能的应用，淘宝根本不可能面对每天亿级的用户访问数量。因此，仅仅因为本能，阿里巴巴一开始就自然生长在这样一个数据的黑洞中，并且被越来越多、越来越密集的数据风暴裹挟。阿里巴巴在大数据方面所做的各种艰苦努力，其实就是力图对抗这种无序和复杂的熵增，从中梳理结构，提炼价值。

这是一个历经磨炼、也卓有成效的长期过程。如书中所提到的，阿里巴巴不仅数据量超宇宙级，而且更是因为业务场景的复杂和多元化，其面对着甚至超过 Google 和 Facebook 的更复杂的难题。大部分时候，阿里巴巴都是在无人区艰难跋涉。每一组功能和逻辑，每一套架构与系统，都与业务和场景息息相关。这个黑洞膨胀之快，以至于大部分时候都是在出现痛点从而刺激了架构升级。换言之，大数据系统——如果我们非要用一个系统去描述的话——其复杂度之高，是几乎不可能在一开始就完整和完美地进行自上而下定义和设计的。从需求→设计→迭代→

升华为理论，在无数次的迭代进化中，我们对大数据的理解才逐渐成形，慢慢能够在将数据黑洞为我所用的抗争中扳回一局。

这个系统生长和进化的过程实际上已经暗暗揭示了阿里巴巴对大数据真髓的理解。大、快、多样性只是表象，大数据的真正价值在于生命性和生态性。阿里巴巴称之为“活数据”。活数据是全本记录、实时驱动决策和迭代，其价值是随着使用场景和方式动态变化的。简单地把数据定义为正/负资产都太简单。数据也不是会枯竭的能源。数据可以被重复使用，并在使用中升值；数据与数据链接可能会像核反应一样产生价值的聚变。数据使用和数据聚变又产生新的数据。活数据的基础设施就需要来承载、管理和促进这个生态体的最大价值实现（以及相应的成本最小化）。丰富的数据形式、多样化的参与角色和动机，以及迥异的计算场景都使得这个系统的复杂度无限升级。阿里巴巴的大数据之路就是在深刻理解这种复杂性的基础上，摸索到了一些重要的秩序和原理，并通过技术架构来验证和夯实。

如果说互联网实现了人人互联和通信，并没有深度地协同计算，那么这样的一个大数据平台和架构就是一张升级的、智能的互联网。这是人类自己设计出来的复杂的信息处理系统，同时也将是真正意义上人类智力大联合的基础设施。这是一个伟大的蓝图，我们敬畏其复杂度和潜能。《大数据之路——阿里巴巴大数据实践》便是阿里巴巴分享对大数据的认知、与世界共创数据智能的重要基石。数据技术及产品部作为阿里巴巴集团的数据中台，一直致力为阿里巴巴集团内、外提供大数据方面的系统服务，承载了阿里巴巴集团大数据梦想至关重要的数据平台建设。相信他们的实践和思考对同行会有很大的启发和借鉴意义。

曾鸣教授

阿里巴巴集团学术委员会主席

湖畔大学教务长

2017年5月

# 目 录

第 1 章 总述	1
----------	---

## 第 1 篇 数据技术篇

第 2 章 日志采集	8
------------	---

2.1 浏览器的页面日志采集	8
----------------	---

2.1.1 页面浏览日志采集流程	9
------------------	---

2.1.2 页面交互日志采集	14
----------------	----

2.1.3 页面日志的服务器端清洗和预处理	15
-----------------------	----

2.2 无线客户端的日志采集	16
----------------	----

2.2.1 页面事件	17
------------	----

2.2.2 控件点击及其他事件	18
-----------------	----

2.2.3 特殊场景	19
------------	----

2.2.4 H5 & Native 日志统一	20
------------------------	----

2.2.5 设备标识	22
------------	----

2.2.6 日志传输	23
------------	----

2.3 日志采集的挑战	24
-------------	----

2.3.1 典型场景	24
------------	----

2.3.2 大促保障	26
------------	----

第 3 章 数据同步	29
------------	----

3.1 数据同步基础	29
------------	----

3.1.1 直连同步	30
------------	----



3.1.2	数据文件同步	30
3.1.3	数据库日志解析同步	31
3.2	阿里数据仓库的同步方式	35
3.2.1	批量数据同步	35
3.2.2	实时数据同步	37
3.3	数据同步遇到的问题与解决方案	39
3.3.1	分库分表的处理	39
3.3.2	高效同步和批量同步	41
3.3.3	增量与全量同步的合并	42
3.3.4	同步性能的处理	43
3.3.5	数据漂移的处理	45
第4章	离线数据开发	48
4.1	数据开发平台	48
4.1.1	统一计算平台	49
4.1.2	统一开发平台	53
4.2	任务调度系统	58
4.2.1	背景	58
4.2.2	介绍	59
4.2.3	特点及应用	65
第5章	实时技术	68
5.1	简介	69
5.2	流式技术架构	71
5.2.1	数据采集	72
5.2.2	数据处理	74
5.2.3	数据存储	78
5.2.4	数据服务	80
5.3	流式数据模型	80
5.3.1	数据分层	80
5.3.2	多流关联	83
5.3.3	维表使用	84
5.4	大促挑战&保障	86

5.4.1 大促特征	86
5.4.2 大促保障	88
<b>第6章 数据服务</b>	<b>91</b>
6.1 服务架构演进	91
6.1.1 DWSOA	92
6.1.2 OpenAPI	93
6.1.3 SmartDQ	94
6.1.4 统一的数据服务层	96
6.2 技术架构	97
6.2.1 SmartDQ	97
6.2.2 iPush	100
6.2.3 Lego	101
6.2.4 uTiming	102
6.3 最佳实践	103
6.3.1 性能	103
6.3.2 稳定性	111
<b>第7章 数据挖掘</b>	<b>116</b>
7.1 数据挖掘概述	116
7.2 数据挖掘算法平台	117
7.3 数据挖掘中台体系	119
7.3.1 挖掘数据中台	120
7.3.2 挖掘算法中台	122
7.4 数据挖掘案例	123
7.4.1 用户画像	123
7.4.2 互联网反作弊	125

## 第2篇 数据模型篇

<b>第8章 大数据领域建模综述</b>	<b>130</b>
8.1 为什么需要数据建模	130
8.2 关系数据库系统和数据仓库	131

8.3	从 OLTP 和 OLAP 系统的区别看模型方法论的选择 .....	132
8.4	典型的数据仓库建模方法论 .....	132
8.4.1	ER 模型 .....	132
8.4.2	维度模型 .....	133
8.4.3	Data Vault 模型 .....	134
8.4.4	Anchor 模型 .....	135
8.5	阿里巴巴数据模型实践综述 .....	136
第 9 章	阿里巴巴数据整合及管理体系 .....	138
9.1	概述 .....	138
9.1.1	定位及价值 .....	139
9.1.2	体系架构 .....	139
9.2	规范定义 .....	140
9.2.1	名词术语 .....	141
9.2.2	指标体系 .....	141
9.3	模型设计 .....	148
9.3.1	指导理论 .....	148
9.3.2	模型层次 .....	148
9.3.3	基本原则 .....	150
9.4	模型实施 .....	152
9.4.1	业界常用的模型实施过程 .....	152
9.4.2	OneData 实施过程 .....	154
第 10 章	维度设计 .....	159
10.1	维度设计基础 .....	159
10.1.1	维度的基本概念 .....	159
10.1.2	维度的基本设计方法 .....	160
10.1.3	维度的层次结构 .....	162
10.1.4	规范化和反规范化 .....	163
10.1.5	一致性维度和交叉探查 .....	165
10.2	维度设计高级主题 .....	166
10.2.1	维度整合 .....	166
10.2.2	水平拆分 .....	169

10.2.3	垂直拆分	170
10.2.4	历史归档	171
10.3	维度变化	172
10.3.1	缓慢变化维	172
10.3.2	快照维表	174
10.3.3	极限存储	175
10.3.4	微型维度	178
10.4	特殊维度	180
10.4.1	递归层次	180
10.4.2	行为维度	184
10.4.3	多值维度	185
10.4.4	多值属性	187
10.4.5	杂项维度	188
第 11 章	事实表设计	190
11.1	事实表基础	190
11.1.1	事实表特性	190
11.1.2	事实表设计原则	191
11.1.3	事实表设计方法	193
11.2	事务事实表	196
11.2.1	设计过程	196
11.2.2	单事务事实表	200
11.2.3	多事务事实表	202
11.2.4	两种事实表对比	206
11.2.5	父子事实的处理方式	208
11.2.6	事实的设计准则	209
11.3	周期快照事实表	210
11.3.1	特性	211
11.3.2	实例	212
11.3.3	注意事项	217
11.4	累积快照事实表	218
11.4.1	设计过程	218
11.4.2	特点	221



11.4.3	特殊处理	223
11.4.4	物理实现	225
11.5	三种事实表的比较	227
11.6	无事实的事实表	228
11.7	聚集型事实表	228
11.7.1	聚集的基本原则	229
11.7.2	聚集的基本步骤	229
11.7.3	阿里公共汇总层	230
11.7.4	聚集补充说明	234

### 第3篇 数据管理篇

第12章	元数据	236
12.1	元数据概述	236
12.1.1	元数据定义	236
12.1.2	元数据价值	237
12.1.3	统一元数据体系建设	238
12.2	元数据应用	239
12.2.1	Data Profile	239
12.2.2	元数据门户	241
12.2.3	应用链路分析	241
12.2.4	数据建模	242
12.2.5	驱动 ETL 开发	243
第13章	计算管理	245
13.1	系统优化	245
13.1.1	HBO	246
13.1.2	CBO	249
13.2	任务优化	256
13.2.1	Map 倾斜	257
13.2.2	Join 倾斜	261
13.2.3	Reduce 倾斜	269

第 14 章 存储和成本管理	275
14.1 数据压缩	275
14.2 数据重分布	276
14.3 存储治理项优化	277
14.4 生命周期管理	278
14.4.1 生命周期管理策略	278
14.4.2 通用的生命周期管理矩阵	280
14.5 数据成本计量	283
14.6 数据使用计费	284
第 15 章 数据质量	285
15.1 数据质量保障原则	285
15.2 数据质量方法概述	287
15.2.1 消费场景知晓	289
15.2.2 数据加工过程卡点校验	292
15.2.3 风险点监控	295
15.2.4 质量衡量	299
<b>第 4 篇 数据应用篇</b>	
第 16 章 数据应用	304
16.1 生意参谋	305
16.1.1 背景概述	305
16.1.2 功能架构与技术能力	307
16.1.3 商家应用实践	310
16.2 对内数据产品平台	313
16.2.1 定位	313
16.2.2 产品建设历程	314
16.2.3 整体架构介绍	317
附录 A 本书插图索引	320

轻松注册成为博文视点社区用户 ([www.broadview.com.cn](http://www.broadview.com.cn)), 扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31438>



# 第1章

## 总述

2014 年，马云提出，“人类正从 IT 时代走向 DT 时代”。如果说在 IT 时代是以自我控制、自我管理为主，那么到了 DT (Data Technology) 时代，则是以服务大众、激发生产力为主。以互联网（或者物联网）、云计算、大数据和人工智能为代表的新技术革命正在渗透至各行各业，悄悄地改变着我们的生活。

在 DT 时代，人们比以往任何时候更能收集到更丰富的数据。IDC 的报告显示：预计到 2020 年，全球数据总量将超过 40ZB（相当于 40 万亿 GB），这一数据量是 2011 年的 22 倍！正在呈“爆炸式”增长的数据，其潜在的巨大价值有待发掘。数据作为一种新的能源，正在发生聚变，变革着我们的生产和生活，催生了当下大数据行业发展热火朝天的盛景。

但是如果不能对这些数据进行有序、有结构地分类组织和存储，如果不能有效利用并发掘它，继而产生价值，那么它同时也成为一场“灾难”。无序、无结构的数据犹如堆积如山的垃圾，给企业带来的是令人咋舌的高额成本。

在阿里巴巴集团内，我们面临的现实情况是：集团数据存储达到



EB 级别，部分单张表每天的数据记录数高达几千亿条；在 2016 年“双 11 购物狂欢节”的 24 小时中，支付金额达到了 1207 亿元人民币，支付峰值高达 12 万笔/秒，下单峰值达 17.5 万笔/秒，媒体直播大屏处理的总数据量高达百亿且所有数据都需要做到实时、准确地对外披露……这些给数据采集、存储和计算都带来了极大的挑战。

在阿里内部，数据工程师每天要面对百万级规模的离线数据处理工作。阿里大数据井喷式的爆发，加大了数据模型、数据研发、数据质量和运维保障工作的难度。

同时，日益丰富的业态，也带来了各种各样、纷繁复杂的数据需求。如何有效地满足来自员工、商家、合作伙伴等多样化的需求，提高他们对数据使用的满意度，是数据服务和数据产品需要面对的挑战。

如何建设高效的数据模型和体系，使数据易用，避免重复建设和数据不一致性，保证数据的规范性；如何提供高效易用的数据开发工具；如何做好数据质量保障；如何有效管理和控制日益增长的存储和计算消耗；如何保证数据服务的稳定，保证其性能；如何设计有效的数据产品高效赋能于外部客户和内部员工……这些都给大数据系统的建设提出了更多复杂的要求。

本书介绍的阿里巴巴大数据系统架构，就是为了满足不断变化的业务需求，同时实现系统的高度扩展性、灵活性以及数据展现的高性能而设计的。

如图 1.1 所示是阿里巴巴大数据系统体系架构图，从图中可以清晰地看到数据体系主要分为数据采集、数据计算、数据服务和数据应用四大层次。

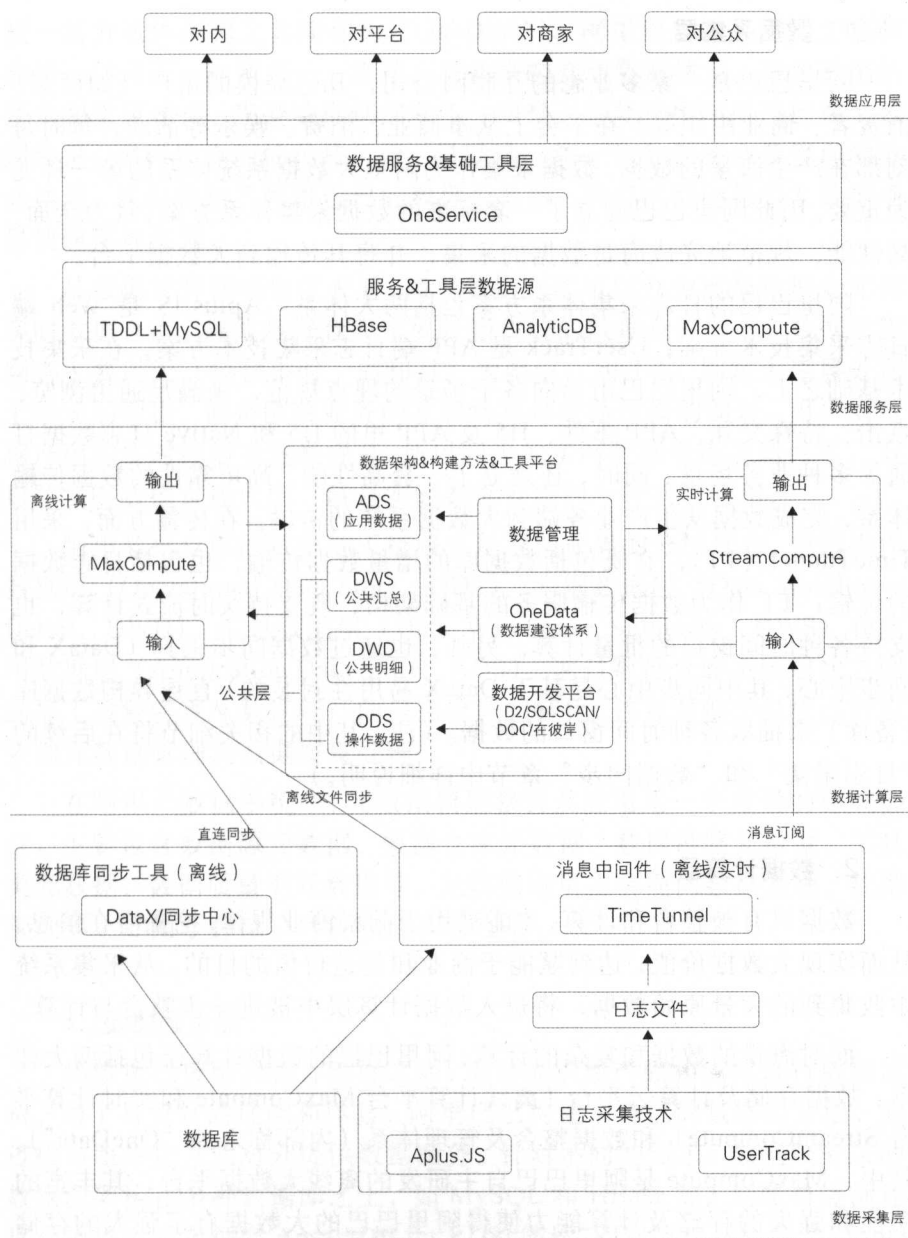


图 1.1 阿里巴巴大数据系统体系架构图

## 1. 数据采集层

阿里巴巴是一家多业态的互联网公司，几亿规模的用户（如商家、消费者、商业组织等）在平台上从事商业、消费、娱乐等活动，每时每刻都在产生海量的数据，数据采集作为阿里大数据系统体系的第一环尤为重要。因此阿里巴巴建立了一套标准的数据采集体系方案，致力全面、高性能、规范地完成海量数据的采集，并将其传输到大数据平台。

阿里巴巴的日志采集体系方案包括两大体系：Aplus.JS 是 Web 端日志采集技术方案；UserTrack 是 APP 端日志采集技术方案。在采集技术基础之上，阿里巴巴用面向各个场景的埋点规范，来满足通用浏览、点击、特殊交互、APP 事件、H5 及 APP 里的 H5 和 Native 日志数据打通等多种业务场景。同时，还建立了一套高性能、高可靠性的数据传输体系，完成数据从生产业务端到大数据系统的传输。在传输方面，采用 TimeTunnel (TT)，它既包括数据库的增量数据传输，也包括日志数据的传输；TT 作为数据传输服务的基础架构，既支持实时流式计算，也支持各种时间窗口的批量计算。另外，也通过数据同步工具（DataX 和同步中心，其中同步中心是基于 DataX 易用性封装的）直连异构数据库（备库）来抽取各种时间窗口的数据。（注：其中的相关细节将在后续的“日志采集”和“数据同步”章节中详细说明。）

## 2. 数据计算层

数据只有被整合和计算，才能被用于洞察商业规律，挖掘潜在信息，从而实现大数据价值，达到赋能于商业和创造价值的目的。从采集系统中收集到的大量原始数据，将进入数据计算层中被进一步整合与计算。

面对海量的数据和复杂的计算，阿里巴巴的数据计算层包括两大体系：数据存储及计算云平台（离线计算平台 MaxCompute 和实时计算平台 StreamCompute）和数据整合及管理体系（内部称之为“OneData”）。其中，MaxCompute 是阿里巴巴自主研发的离线大数据平台，其丰富的功能和强大的存储及计算能力使得阿里巴巴的大数据有了强大的存储和计算引擎；StreamCompute 是阿里巴巴自主研发的流式大数据平台，在内部较好地支持了阿里巴巴流式计算需求；OneData 是数据整合及管理的方法体系和工具（注：为方便内部工作及沟通，在阿里内部将这一

统一的方法体系和工具简称为“OneData”), 阿里巴巴的大数据工程师在这一体系下, 构建统一、规范、可共享的全域数据体系, 避免数据的冗余和重复建设, 规避数据烟囱和不一致性, 充分发挥阿里巴巴在大数据海量、多样性方面的独特优势。借助这一统一化数据整合及管理的方法体系, 我们构建了阿里巴巴的数据公共层, 并可以帮助相似大数据项目快速落地实现。

从数据计算频率角度来看, 阿里数据仓库可以分为离线数据仓库和实时数据仓库。离线数据仓库主要是指传统的数据仓库概念, 数据计算频率主要以天(包含小时、周和月)为单位; 如 T-1, 则每天凌晨处理上一天的数据。但是随着业务的发展特别是交易过程的缩短, 用户对数据产出的实时性要求逐渐提高, 所以阿里的实时数据仓库应运而生。“双11”实时数据直播大屏, 就是实时数据仓库的一种典型应用。

阿里数据仓库的数据加工链路也是遵循业界的分层理念, 包括操作数据层(Operational Data Store, ODS)、明细数据层(Data Warehouse Detail, DWD)、汇总数据层(Data Warehouse Summary, DWS)和应用数据层(Application Data Store, ADS)。通过数据仓库不同层次之间的加工过程实现从数据资产向信息资产的转化, 并且对整个过程进行有效的元数据管理及数据质量处理。

在阿里大数据系统中, 元数据模型整合及应用是一个重要的组成部分, 主要包含数据源元数据、数据仓库元数据、数据链路元数据、工具类元数据、数据质量类元数据等。元数据应用主要面向数据发现、数据管理等, 如用于存储、计算和成本管理等。

### 3. 数据服务层

当数据已被整合和计算好之后, 需要提供给产品和应用进行数据消费。为了有更好的性能和体验, 阿里巴巴构建了自己的数据服务层, 通过接口服务化方式对外提供数据服务。针对不同的需求, 数据服务层的数据源架构在多种数据库之上, 如 MySQL 和 HBase 等。后续将逐渐迁移至阿里云云数据库 ApsaraDB for RDS(简称“RDS”)和表格存储(Table Store)等。

数据服务可以使应用对底层数据存储透明, 将海量数据方便高效地

开放给集团内部各应用使用。现在，数据服务每天拥有几十亿的数据调用量，如何在性能、稳定性、扩展性等方面更好地服务于用户；如何满足应用各种复杂的数据服务需求；如何保证“双 11”媒体大屏数据服务接口的高可用……随着业务的发展，需求越来越复杂，因此数据服务也在不断地前进。

数据服务层对外提供数据服务主要是通过统一的数据服务平台（为方便阅读，简称为“OneService”）。OneService 以数据仓库整合计算好的数据作为数据源，对外通过接口的方式提供数据服务，主要提供简单数据查询服务、复杂数据查询服务（承接集团用户识别、用户画像等复杂数据查询服务）和实时数据推送服务三大特色数据服务。

#### 4. 数据应用层

数据已经准备好，需要通过合适的应用提供给用户，让数据最大化地发挥价值。阿里对数据的应用表现在各个方面，如搜索、推荐、广告、金融、信用、保险、文娱、物流等。商家，阿里内部的搜索、推荐、广告、金融等平台，阿里内部的运营和管理人员等，都是数据应用方；ISV、研究机构和社会组织等也可以利用阿里开放的数据能力和技术。

阿里巴巴基于数据的应用产品有很多，本书选择了服务于阿里内部员工的阿里数据平台和服务于商家的对外数据产品——生意参谋进行基础性介绍。其他数据应用不再赘述。对内，阿里数据平台产品主要有实时数据监控、自助式的数据网站或产品构建的数据小站、宏观决策分析支撑平台、对象分析工具、行业数据分析门户、流量分析平台等。

我们相信，数据作为新能源，为产业注入的变革是显而易见的。我们对数据新能源的探索也不仅仅停留在狭义的技术、服务和应用上。我们正在挖掘大数据更深层次的价值，为社会经济和民生基础建设等提供创新方法。

注：本书中出现的专有名词、专业术语、产品名称、软件项目名称、工具名称等，是淘宝（中国）软件有限公司内部项目的惯用词语，如与第三方名称雷同，实属巧合。

# 第1篇

## 数据技术篇

- 第2章 日志采集
- 第3章 数据同步
- 第4章 离线数据开发
- 第5章 实时技术
- 第6章 数据服务
- 第7章 数据挖掘

## 第2章

# 日志采集

数据采集作为阿里大数据系统体系的第一环尤为重要。因此阿里巴巴建立了一套标准的数据采集体系方案，致力全面、高性能、规范地完成海量数据的采集，并将其传输到大数据平台。本章主要介绍数据采集中的日志采集部分。

阿里巴巴的日志采集体系方案包括两大体系：Aplus.JS 是 Web 端（基于浏览器）日志采集技术方案；UserTrack 是 APP 端（无线客户端）日志采集技术方案。

本章从浏览器的页面日志采集、无线客户端的日志采集以及我们遇到的日志采集挑战三块内容来阐述阿里巴巴的日志采集经验。

### 2.1 浏览器的页面日志采集

浏览器的页面型产品/服务的日志采集可分为如下两大类。

(1) 页面浏览（展现）日志采集。顾名思义，页面浏览日志是指当一个页面被浏览器加载呈现时采集的日志。此类日志是最基础的互联网



日志，也是目前所有互联网产品的两大基本指标：页面浏览量（Page View，PV）和访客数（Unique Visitors，UV）的统计基础。页面浏览日志是目前成熟度和完备度最高，同时也是最具挑战性的日志采集任务，我们将重点讲述此类日志的采集。

（2）页面交互日志采集。当页面加载和渲染完成之后，用户可以在页面上执行各类操作。随着互联网前端技术的不断发展，用户可在浏览器内与网页进行的互动已经丰富到只有想不到没有做不到的程度，互动设计都要求采集用户的互动行为数据，以便通过量化获知用户的兴趣点或者体验优化点。交互日志采集就是为此类业务场景而生的。

除此之外，还有一些专门针对某些特定统计场合的日志采集需求，如专门采集特定媒体在页面被曝光状态的曝光日志、用户在线状态的实时监测等，但在基本原理上都脱胎于上述两大类。限于篇幅，此内容在本书中就不予展开介绍了。

### 2.1.1 页面浏览日志采集流程

网站页面是互联网服务的基本载体，即使在如今传统互联网形态逐渐让位于移动互联网的背景下，HTML 页面依旧是最普遍的业务形态，对于以网页为基本展现形式的互联网产品和服务，衡量其业务水平的基本指标是网页浏览量（PV）和访客数（UV）。为此，我们需要采集页面被浏览器加载展现的记录，这是最原始的互联网日志采集需求，也是一切互联网数据分析得以展开的基础和前提。

目前典型的网页访问过程是以浏览器请求、服务器响应并返回所请求的内容（大多以 HTML 文档的形式）这种模式进行的，浏览器和服务器之间的通信普遍遵守 HTTP 协议（超文本传输协议，目前以 HTTP 1.1 为主，逐渐向最新的 HTTP 2.0 过渡）。浏览器发起的请求被称为 HTTP 请求（HTTP Request），服务器的返回则被称为 HTTP 响应（HTTP Response）。

我们以用户访问淘宝首页（[www.taobao.com](http://www.taobao.com)）为例，一次典型的页面访问过程描述如图 2.1 所示。



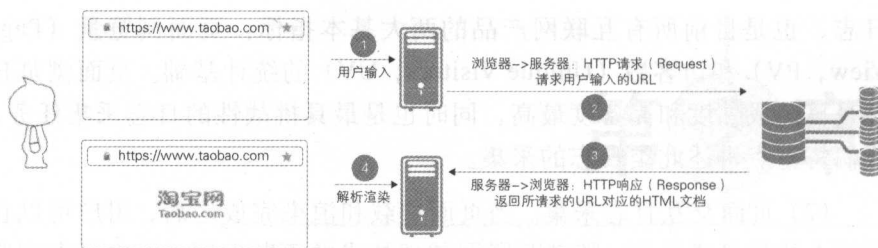


图 2.1 一次典型的互联网页面请求-响应过程

(1) 用户在浏览器内点击淘宝首页链接（或在地址栏中输入 `www.taobao.com` 并回车）。

(2) 浏览器向淘宝服务器发起 HTTP 请求。在本例子中，用户可以看见的内容只是显示于浏览器地址栏内的 `http://www.taobao.com`，而浏览器在执行时，会解析用户请求并按照 HTTP 协议中约定的格式将其转化为一个 HTTP 请求发送出去。

按照 HTTP 协议，一个标准的 HTTP 请求由如下三部分构成。

- 请求行 (HTTP Request Line)。请求行内有三个要素，分别是请求方法、所请求资源的 URL 以及 HTTP 协议版本号。在本例子中，这三个要素分别是 GET、`http://www.taobao.com/` 以及 HTTP 1.1，对于我们所讨论的话题，记住请求行内最重要的信息是这个 URL 就可以了。
- 请求报头 (HTTP Message Header)。请求报头是浏览器在请求时向服务器提交的附加信息，请求报头一般会附加很多内容项（每项内容被称为一个头域 (Header Field)，在不引起混淆的情况下，往往将 Header Field 简称为 Header）。需要注意的是，如果用户在本次页面访问之前已经到访过网站或者已经登录，则一般都会在请求头中附加一个或多个被称为 Cookie 的数据项，其中记录了用户上一次访问时的状态或者身份信息，我们只需理解浏览器在发起请求时会带上一个标明用户身份的 Cookie 即可。
- 请求正文 (HTTP Message Body)。这一部分是可选的，一般而言，HTTP 请求的正文都是空的，可以忽略。

(3) 服务器接收并解析请求。服务器端的业务处理模块按业务逻辑处理本次请求并按照 HTTP 协议规定的格式,将处理结果以 HTTP 响应形式发回浏览器。

与 HTTP 请求相对应,一个标准的 HTTP 响应也由三部分构成。

- 状态行。状态行标识了服务器对于此次 HTTP 请求的处理结果。状态行内的主要内容是一个由三位数字构成的状态码,我们最熟知的两个状态码分别是代表成功响应的 200 (OK) 和代表所请求的资源在服务器端没有被找到的 404 (Not Found)。
- 响应报头。服务器在执行响应时,同样可以附加一些数据项,这些数据项将在浏览器端被读取和使用。事实上,在大多数页面和应用中,响应报头内的内容在确保页面正确显示和业务正常进行方面都发挥着至关重要的作用。其中最重要的一类 Header 即上面所提到的 Cookie,浏览器所记录的 Cookie,其实是由服务器在响应报头内指令浏览器记录的。举个例子,如果用户在页面登录,则服务器会在登录请求的响应报头内指示浏览器新增一个名为 userid 的 Cookie 项,其中记录了登录用户的 id。如此一来,当用户随后再次访问该网站时,浏览器将自动在请求报头内附加这个 Cookie,服务器由此即可得知本次请求对应的用户到底是谁;如果服务器发现浏览器在请求时传递过来的 Cookie 有缺失、错误或者需要更新,则会在响应报头内指令浏览器增加或更新对应的 Cookie。
- 响应正文。和请求正文一样,这一部分在协议中也被定义为可选部分,但对于大多数 HTTP 响应而言,这一部分都是非空的,浏览器请求的文档、图片、脚本等,其实就是被包装在正文内返回浏览器的。在本例子中,服务器会将淘宝首页对应的 HTML 文档封装在正文内。

(4) 浏览器接收到服务器的响应内容,并将其按照文档规范展现给用户,从而完成一次请求。在本例子中,浏览器请求淘宝首页,服务器返回对应的 HTML 文档,浏览器即按照 HTML 文档规范解析文档并将整个页面渲染在屏幕上。

上面描述了一次典型的网页浏览过程，如果需要记录这次浏览行为，则采集日志的动作必然是附加在上述四个步骤中的某一环节内完成的。在第一步和第二步，用户的请求尚未抵达服务器；而直到第三步完成，我们也只能认为服务器处理了请求，不能保证浏览器能够正确地解析和渲染页面，尚不能确保用户已确实打开页面，因此在前三步是无法采集用户的浏览日志的。那么采集日志的动作，需要在第四步，也就是浏览器开始解析文档时才能进行。根据前文所述，可以很自然地得出在这一模式下最直接的日志采集思路：在 HTML 文档内的适当位置增加一个日志采集节点，当浏览器解析到这个节点时，将自动触发一个特定的 HTTP 请求到日志采集服务器。如此一来，当日志采集服务器接收到这个请求时，就可以确定浏览器已经成功地接收和打开了页面。这就是目前几乎所有互联网网站页面浏览日志采集的基本原理，而业界的各类网页日志采集的解决方案只是在实施的细节、自动采集内容的广度以及部署的便利性上有所不同。

目前阿里巴巴采用的页面浏览日志采集方案的流程框架如图 2.2 所示。在图 2.2 所示的页面浏览日志采集过程中，所涉及的日志相关的几个主要过程简单介绍如下：

(1) 客户端日志采集。日志采集工作一般由一小段被植入页面 HTML 文档内的 JavaScript 脚本来执行。采集脚本被浏览器加载解析后执行，在执行时采集当前页面参数、浏览行为的上下文信息（如读取用户访问当前页面时的上一步页面）以及一些运行环境信息（如当前的浏览器和分辨率等）。在 HTML 文档内植入日志采集脚本的动作可以由业务服务器在响应业务请求时动态执行，也可以在开发页面时由开发人员手动植入。在阿里巴巴，这两种方式均有采用，其中前一种方式的占比较高，这一点与业界的普遍状况有所不同。图 2.2 中的第三、四步描述了阿里业务服务器端植入日志采集脚本的过程。

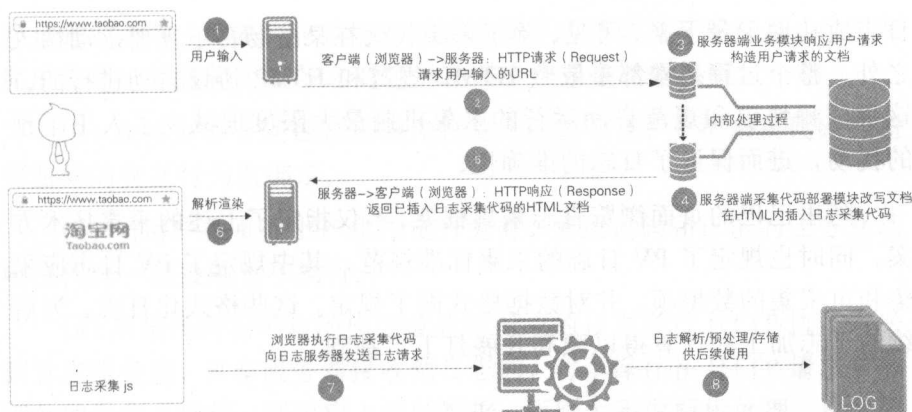


图 2.2 阿里巴巴页面浏览日志采集方案流程框架

(2) 客户端日志发送。采集脚本执行时，会向日志服务器发起一个日志请求，以将采集到的数据发送到日志服务器。在大多数情况下，采集完成之后会立即执行发送；但在个别场景下，日志采集之后可能会经过一段时间的延迟才被发出。日志采集和发送模块一般会集成在同一个 JavaScript 脚本文件内，且通过互联网浏览器必然支持的 HTTP 协议与日志服务器通信，采集到的日志信息一般以 URL 参数形式放在 HTTP 日志请求的请求行内。

(3) 服务器端日志收集。日志服务器接收到客户端发来的日志请求后，一般会立即向浏览器发回一个请求成功的响应，以免对页面的正常加载造成影响；同时，日志服务器的日志收集模块会将日志请求内容写入一个日志缓冲区内，完成此条浏览日志的收集。

(4) 服务器端日志解析存档。服务器接收到的浏览日志进入缓冲区后，会被一段专门的日志处理程序顺序读出并按照约定的日志处理逻辑解析。由日志采集脚本记录在日志请求行内的参数，将在这个环节被解析（有时候伴随着转义和解码）出来，转存入标准的日志文件中并注入实时消息通道内供其他后端程序读取和进一步加工处理。

经过采集—发送—收集—解析存档四个步骤，我们将一次页面浏览

日志成功地记录下来。可见，除了采集代码在某些场合下需要手动植入之外，整个过程基本都是依照 HTML 规范和 HTTP 协议自动进行的，这种依赖协议和规范自动运行的采集机制最大限度地减少了人工干预的扰动，进而保证了日志的准确性。

阿里巴巴的页面浏览日志采集框架，不仅指定了上述的采集技术方案，同时也规定了 PV 日志的采集标准规范，其中规定了 PV 日志应采集和可采集的数据项，并对数据格式做了规定。这些格式化日志，为后续的日志加工和计算得以顺利开展打下了基础。

## 2.1.2 页面交互日志采集

PV 日志的采集解决了页面流量和流量来源统计的问题，但随着互联网业务的发展，仅了解用户到访过的页面和访问路径，已经远远不能满足用户细分研究的需求。在很多场合下，需要了解用户在访问某个页面时具体的互动行为特征，比如鼠标或输入焦点的移动变化（代表用户关注内容的变化）、对某些页面交互的反应（可借此判断用户是否对某些页面元素发生认知困难）等。因为这些行为往往并不触发浏览器加载新页面，所以无法通过常规的 PV 日志采集方法来收集。在阿里巴巴，通过一套名为“黄金令箭”的采集方案来解决交互日志的采集问题。

因为终端类型、页面内容、交互方式和用户实际行为的千变万化不可预估，交互日志的采集和 PV 日志的采集不同，无法规定统一的采集内容（例如，活动页面的游戏交互和淘宝购物车页面的功能交互两者相比，所需记录的行为类型、行为数据以及数据的结构化程度都截然不同），呈现出高度自定义的业务特征。与之相适应，在阿里巴巴的日志采集实践中，交互日志的采集（即“黄金令箭”）是以技术服务的形式呈现的。

具体而言，“黄金令箭”是一个开放的基于 HTTP 协议的日志服务，需要采集交互日志的业务（下文简称“业务方”），经过如下步骤即可将自助采集的交互日志发送到日志服务器。

（1）业务方在“黄金令箭”的元数据管理界面依次注册需要采集交

互日志的业务、具体的业务场景以及场景下的具体交互采集点，在注册完成之后，系统将生成与之对应的交互日志采集代码模板。

(2) 业务方将交互日志采集代码植入目标页面，并将采集代码与需要监测的交互行为做绑定。

(3) 当用户在页面上产生指定行为时，采集代码和正常的业务互动响应代码一起被触发和执行。

(4) 采集代码在采集动作完成后将对应的日志通过 HTTP 协议发送到日志服务器，日志服务器接收到日志后，对于保存在 HTTP 请求参数部分的自定义数据，即用户上传的数据，原则上不做解析处理，只做简单的转储。

经过上述步骤采集到日志服务器的业务随后可被业务方按需自行解析处理，并可与正常的 PV 日志做关联运算。

### 2.1.3 页面日志的服务器端清洗和预处理

上面介绍了阿里巴巴的两类浏览器页面日志的采集方案，并粗略介绍了日志到达日志服务器之后的解析处理。但在大部分场合下，经过上述解析处理之后的日志并不直接提供给下游使用。基于如下几个原因，在对时效要求较宽松的应用场合下，一般还需要进行相应的离线预处理。

(1) 识别流量攻击、网络爬虫和流量作弊（虚假流量）。页面日志是互联网分析和大数据应用的基础源数据，在实际应用中，往往存在占一定比例的虚假或者恶意流量日志，导致日志相关指标的统计发生偏差或明显谬误。为此，需要对所采集的日志进行合法性校验，依托算法识别非正常的流量并归纳出对应的过滤规则集加以滤除。这是一个长期而艰苦的对抗过程。

(2) 数据缺项补正。为了便利后续的日志应用和保证基本的数据统计口径一致，在大多数情况下，需要对日志中的一些公用且重要的数据项做取值归一、标准化处理或反向补正。反向补正，即根据新日志对稍早收集的日志中的个别数据项做回补或修订（例如，在用户登录后，对



登录前页面日志做身份信息的回补)。

(3) 无效数据剔除。在某些情况下,因业务变更或配置不当,在采集到的日志中会存在一些无意义、已经失效或者冗余的数据项。这些数据项不仅消耗存储空间和运算能力,而且在偶然的情况下还可能干扰正常计算的进行。为了避免此类异常的发生,需要定时检查配置并依照配置将此类数据项剔除。

(4) 日志隔离分发。基于数据安全或者业务特性的考虑,某些日志在进入公共数据环境之前需要做隔离。

原始日志经过上述的清洗、修正,并结构化变形处理之后,Web 页面日志的采集流程就算完成了。此时的日志已经具备了结构化或者半结构化的特征,可以方便地被关系型数据库装载和使用。

## 2.2 无线客户端的日志采集

众所周知,日志采集多是为了进行后续的数据分析。移动端的数据采集,一是为了服务于开发者,协助开发者分析各类设备信息;二是为了帮助各 APP 更好地了解自己的用户,了解用户在 APP 上的各类行为,帮助各应用不断进行优化,提升用户体验。

无线客户端的日志采集采用采集 SDK 来完成,在阿里巴巴内部,多使用名为 UserTrack 的 SDK 来进行无线客户端的日志采集。无线客户端的日志采集和浏览器的日志采集方式有所不同,移动端的日志采集根据不同的用户行为分成不同的事件,“事件”为无线客户端日志行为的最小单位。基于常规的分析,UserTrack (UT) 把事件分成了几类,常用的包括页面事件(同前述的页面浏览)和控件点击事件(同前述的页面交互)等。

对事件进行分类的原因,除了不同事件的日志触发时机、日志内容和实现方式有差异之外,另一方面是为了更好地完成数据分析。在常见的业务分析中,往往较多地涉及某类事件,而非全部事件;故为了降低

后续处理的复杂性,对事件进行分类尤为重要。要更好地进行日志数据分析,涉及很多方面的内容,如需要处理 Hybrid 应用,实现 H5 和 Native 日志的统一;又如识别设备,保证同一设备上各应用获取到的设备信息是唯一的。除此之外,对于采集到的数据如何上传,以及后续又如何合理处理等,每个过程都值得我们进行深入的研究和探索。

### 2.2.1 页面事件

从实现方法上说,日志采集 SDK 对于不同事件的实现,大致是类似的;只是对于通用的用户行为,抽象出来一些通用的接口方法。我们把常用的行为类别单独列出来,作为单独的事件来处理,如本节要讲的页面事件(页面浏览行为)。每条页面事件日志记录三类信息:①设备及用户的基本信息;②被访问页面的信息,这里主要是一些业务参数(如商品详情页的商品 ID、所属的店铺等);③访问基本路径(如页面来源、来源的来源等),用于还原用户完整的访问行为。

对于页面事件,不同的 SDK 有不同的实现,有些采集 SDK 选择在页面创建时即发送日志。结合业务分析,UT 提供了页面事件的无痕埋点,即无须开发者进行任何编码即可实现。限于篇幅,本处主要讲一下手动模式的埋点。UT 提供了两个接口,分别在页面展现和页面退出时调用。以进入手机淘宝的某店铺详情页来举例,当进入该店铺详情页时,调用页面展现的接口,该接口会记录页面进入时的一些状态信息,但此时不发送日志;当从该店铺详情页离开时(可能是在店铺详情页上点击某个商品到了对应的商品详情页,也可能是退出了手机淘宝,抑或是点击返回,返回到了之前的一个页面),调用页面退出的接口,该接口会发送日志。除了基础的两个接口外,还提供了添加页面扩展信息的接口;在页面离开前,使用该接口提供的方法给页面添加相关参数,比如给店铺详情页添加店铺 ID、店铺类别(天猫店铺或淘宝店铺)等。

显然,上述三个接口方法必须配合使用,即页面展现和页面退出方法必须成对使用,而页面扩展信息的接口必须在使用页面展现和页面退出方法的前提下使用。再来说说,为什么不在页面进入时就发送日志,而是在页面离开时才发送日志呢?可以思考一下:基于浏览器的日志采



集，在每次页面进入时就实现采集日志的发送，每个页面停留时长的计算一直困扰着分析师；而无线客户端的日志采集，在页面离开时发送日志，此时页面停留时长就是天然自带的准确值了。还可以进一步思考，还有什么其他的优势呢？

上述三个方法是采集 SDK 提供的页面事件采集的基础方法；除此之外，为了平衡采集、计算和分析的成本，在部分场景下我们选择采集更多的信息来减少计算及分析的成本。于是，UT 提供了透传参数功能。所谓透传参数，即把当前页面的某些信息，传递到下一个页面甚至下一个页面的日志中。举个例子，在手机淘宝首页搜索“连衣裙”，进入搜索 list 页面，然后点击某个商品进入商品 A 详情页。如果需要分析“连衣裙”这个关键词，或者商品 A 的来源搜索词，此时就需要把“连衣裙”这个关键词带入到搜索 list 页面日志、商品 A 详情页日志中，这样一来，分析搜索词的效果就显而易见了。在阿里系内，使用 SPM (Super Position Model, 超级位置模型) 进行来源去向的追踪，在无线客户端也同样使用 SPM，SPM 信息就可以通过透传机制带入到下一步甚至下下一步的浏览页面，这样整个用户行为路径还原就轻松实现了。

## 2.2.2 控件点击及其他事件

为了和基于浏览器客户端的日志采集做比较，我们暂且把除了页面事件外的各类事件放到一起来讲。

和浏览器客户端的日志采集一样，交互日志的采集无法规定统一的采集内容，交互类的行为呈现出高度自定义的业务特征。与之相适应，在阿里巴巴的无线客户端日志采集实践中，将交互日志采集从页面事件采集中剥离出来，这就是控件点击事件和其他事件。

先来说说控件点击事件。控件点击事件比页面事件要简单得多，首先，它和页面事件一样，记录了基本的设备信息、用户信息；其次，它记录了控件所在页面名称、控件名称、控件的业务参数等。由于控件点击事件的逻辑简单得多，就是操作页面上的某个控件，因此只需把相关基础信息告诉采集 SDK 即可。

再来说说其他事件。所谓其他事件，就是用户可以根据业务场景需求，使用自定义事件来采集相关信息。从某种程度上说，它几乎能满足用户的所有需求，包括页面事件和控件点击事件，只是若采用通用的页面事件埋点方法，UT 会帮助实现一些额外的功能（如上个页面的信息）。UT 提供了一个自定义埋点类，其包括：①事件名称；②事件时长；③事件所携带的属性；④事件对应的页面。当然，具体实现什么功能，需要带哪些内容，各个采集 SDK 可以自行决定。

除了上述这些需要应用开发者触发的日志采集接口方法外，UT 还提供了一些默认的日志采集方法，比如可以自动捕获应用崩溃，并且产生一条日志记录崩溃相关信息。类似的日志采集方法还有很多，比如应用的退出、页面的前后台切换等。诸如一些和业务信息不是非常相关，但又对分析起很大作用的日志采集，就完全没有必要让应用开发者去触发埋点了。

### 2.2.3 特殊场景

上述场景均为一个行为产生一条日志，如一次浏览、一次点击等。如此用来处理普通的业务是足够的，但对于阿里巴巴巨大的业务体量来说，为了平衡日志大小，减小流量消耗、采集服务器压力、网络传输压力等，采集 SDK 提供了聚合功能，对某些场景如曝光或一些性能技术类日志，我们提倡在客户端对这类日志进行适当聚合，以减少对日志采集服务器端的请求，适当减小日志大小。总体思路就是每个曝光的元素一般都属于一个页面，利用页面的生命周期来实现适当的聚合及确定发送时机。拿曝光日志来举例，若一个商品的一次曝光就对应一条日志的话，那么在搜索结果页的一次滚屏浏览过程中将产生几十条甚至上百条日志，从下游使用及分析的角度来说，其实只是想知道哪些内容被曝光，此时为了平衡业务需求及减少全链路资源消耗，采集 SDK 提供了本地聚合功能，在客户端对这类日志进行聚合，上传聚合后的日志到采集服务器即可。同时对于一些只需要计数，而不需要知道具体内容的场景，如需要分析某些接口的调用次数，此功能就更加凸显出其作用了。

区别于浏览器的页面访问，在无线客户端用户的访问行为路径存在

明显的回退行为（如点击回退按钮、各种滑屏等），在进行业务分析时，回退同样作为特殊场景而存在。例如，“双11”主会场页→女装分会场→女装店铺A→回退到女装分会场→女装店铺B，在这个访问路径中，若只是按照普通的路径来处理，则会认为第二次浏览的女装分会场来源为店铺A，就会把女装分会场的一次浏览效果记为女装店铺A带来的。倘若这样处理，就会发现类似的活动承接页其来源有一大部分均为各类详情页（店铺详情页/商品详情页），这必然干扰到分析。所以针对这种场景，我们做了特殊的设计，利用页面的生命周期，识别页面的复用，配合栈的深度来识别是否是回退行为。

如上列举了两个比较典型的特殊场景，随着业务的不断发展，业务的复杂性不断提高，采集需要处理的特殊场景也层出不穷，限于篇幅，此处不再一一展开介绍。

## 2.2.4 H5 & Native 日志统一

简单来说，APP分为两种：一种是纯Native APP；一种是既有Native，又有H5页面嵌入的APP，即Hybrid APP。当前，纯Native APP已经非常少了，一般都是Hybrid APP。Native页面采用采集SDK进行日志采集，H5页面一般采用基于浏览器的页面日志采集方式进行采集。在当前的实践中，由于采集方式的不同，采集到的内容及采集服务器均分离开。若需要进行完整的数据分析，就需要将两类日志在数据处理时进行关联，而就算不考虑处理成本，在很多情况下，Native和H5互跳，即使关联也无法还原用户路径，数据丢失严重。对于产品经理以及运营、管理、数据分析人员而言，在不同的终端采用不同的方案采集日志，以不同的算法来做日志统计，忍受多端之间的数据隔离，并对由此导致的多样数据口径进行整理分析和解释，已经是越来越不能容忍的切身之痛。考虑到后续日志数据处理的便捷性、计算成本、数据的合理性及准确性，我们需要对Native和H5日志进行统一处理（见图2.3）。

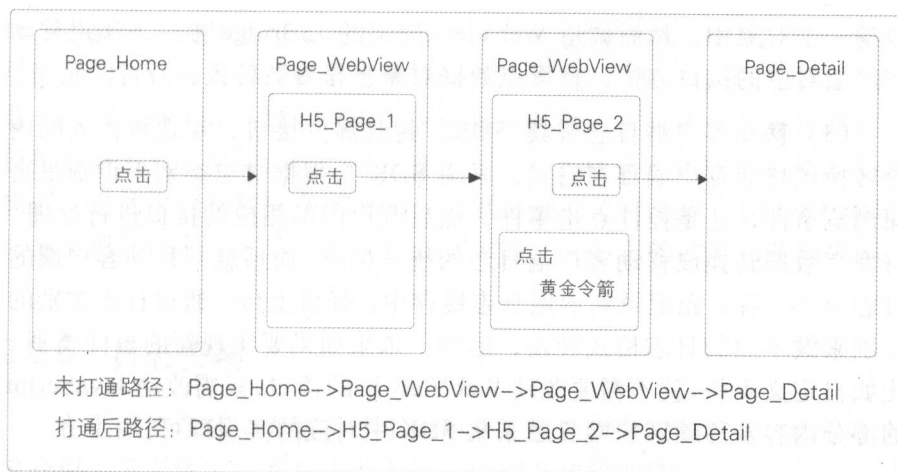


图 2.3 Native 和 H5 日志桥接示意图

要想实现 Native 和 H5 日志的统一处理,就需要对 Hybrid 日志有统一的方案。简单的思路就是首先将两类日志进行归一。用什么方式把两类日志归一呢?是把 Native 日志向 H5 日志归,还是把 H5 日志归到 Native 日志呢?其实两条路均可以实现,没有绝对的答案。选择时可以自行斟酌,在阿里巴巴集团,分别考虑两条路的优缺点,考虑到两种日志采集方式的特点以及关注点,我们选择 Native 部署采集 SDK 的方式。

原因有二:一是采用采集 SDK 可以采集到更多的设备相关数据,这在移动端的数据分析中尤为重要;二是采集 SDK 处理日志,会先在本机缓存,而后借机上传,在网络状况不佳时延迟上报,保证数据不丢失。基于这两点,我们选择将 H5 日志归到 Native 日志。

具体的流程如下:

(1) H5 页面浏览和页面交互的数据,在执行时通过加载日志采集的 JavaScript 脚本,采集当前页面参数,包括浏览行为的上下文信息以及一些运行环境信息。在 APP 中打开 H5 页面和在浏览器中的处理完全一样,在前端页面的开发中无须做任何特殊的处理,只需在页面开发时手动植入日志采集的 JavaScript 脚本即可。

(2) 在浏览器日志采集的 JavaScript 脚本中实现将所采集的数据打

包到一个对象中，然后调用 WebView 框架的 JSBridge 接口，调用移动客户端对应的接口方法，将埋点数据对象当作参数传入。

(3) 移动客户端日志采集 SDK，封装提供接口，实现将传入的内容转换成移动客户端日志格式。采集 SDK 会根据日志类别来识别是页面浏览事件，还是控件点击事件，然后调用内部相应的接口进行处理，将埋点数据转换成移动客户端日志的统一格式。而后就同移动客户端的日志处理一样，先记录到本地日志缓存中，择机上传。通过日志类别的识别来做不同的日志格式转换，这样，未来如果要实现新的事件类别，比如自定义事件，就不需要改动 WebView 层的接口，只需改动 JavaScript 的部分内容及移动客户端日志采集 SDK 中对应的实现即可。

基于这种统一的方案，为后续构建完整的用户行为路径还原树打下了基础。当然，此方案也有其局限性，必须要浏览器采集 JavaScript、WebView、客户端采集 SDK 的配合。而往往很多时候业务并不希望做任何调整，更多的是希望减少依赖，所以在这方面我们也在寻求新的突破。

## 2.2.5 设备标识

正如本章开头所说的，所有互联网产品的两大基本指标是页面浏览量 (Page View, PV) 和访客数 (Unique Visitors, UV)。关于 UV，对于登录用户，可以使用用户 ID 来进行唯一标识，但是很多日志行为并不要求用户登录，这就导致在很多情况下采集上来的日志都没有用户 ID。PC 端一般使用 Cookie 信息来作为设备的唯一信息，对于 APP 来说，我们就要想办法获取到能够唯一标识设备的信息。

历史上，MEI、IMSI、MAC、苹果禁用之前的 UDID，曾经都可以用，如果它们之中有一个是靠谱的话，那么设备唯一标识就简单了。但事实上，随着用户的自我保护意识加强以及各系统升级，很多基本的设备信息获取都不再那么容易。苹果 UDID 禁用，IDFA、IMEI、IMSI 做了权限控制，Android 新系统也对设备信息的获取做了权限控制。

对于只有单 APP 的公司来说，设备唯一标识不是需要攻克的难题，但对于像阿里巴巴这样拥有众多 APP 的公司来说，设备唯一标识就显

得尤为重要。阿里巴巴集团无线设备唯一标识使用 UTDID，它当然有很清晰的责任和目标，就是每台设备一个 ID 作为唯一标识。UTDID 随着 iOS 和 Android 系统对权限控制的不断升级，对方案做了多次调整，包括存储方式、存储位置、共享方式等，以及和服务器端的配合，其生成方式也使用一套较完备的算法。但就目前的进展来说，UTDID 还未完全实现其使命。在这方面，欢迎有思路、有想法的读者和我们联系。

## 2.2.6 日志传输

在上面的章节中大概讲述了如何从无线客户端采集日志，本节将简单介绍一下无线客户端日志的上传、压缩及传输的特殊性。

无线客户端日志的上传，不是产生一条日志上传一条，而是无线客户端产生日志后，先存储在客户端本地，然后再伺机上传。所谓伺机，就需要有数据分析的支持，如在启动后、使用过程中、切换到后台时这些场景下分别多久触发一次上传动作。当然单纯地靠间隔时间来决定上传动作是不够的，还需要考虑日志的大小及合理性（如单条日志超大，很可能就是错误日志）。另外，还需要考虑上传时网络的耗时，来决定是否要调整上传机制。

客户端数据上传时是向服务器发送 POST 请求，服务器端处理上传请求，对请求进行相关校验，将数据追加到本地文件中进行存储，存储方式使用 Nginx 的 access\_log，access\_log 的切分维度为天，即当天接收的日志存储到当天的日志文件中。考虑到后续的数据处理，以及特殊时期不同日志的保障级别，还对日志进行了分流。比如阿里巴巴集团的 Adash（无线日志服务器端处理程序），根据应用及事件类型对每日高达数千亿的日志进行了分流。分流的好处显而易见，如“双 11”时，日常数千亿的日志可能冲高到万亿，此时服务器及后续的数据计算压力就非常大了；而对于重要的数据计算来说，很可能只需要页面事件及控件点击事件即可，此时就可以适当地释放其他类型日志的资源来处理更重要的页面事件及控件点击事件。

从客户端用户行为，到日志采集服务器的日志，整个日志采集的过



程就算结束了。那么日志采集服务器的日志怎么给到下游呢？方法有很多，阿里巴巴集团主要使用消息队列（TimeTunnel, TT）来实现从日志采集服务器到数据计算的 MaxCompute。简单来讲，就是 TT 将消息收集功能部署到日志采集服务器上进行消息的收集，而后续的应用可以是实时的应用实时来订阅 TT 收集到的消息，进行实时计算，也可以是离线的应用定时来获取消息，完成离线的计算。有关消息队列，以及日志数据的统计计算等细节内容，将在后续章节中进行详细讲述。

## 2.3 日志采集的挑战

对于目前的互联网行业而言，互联网日志早已跨越初级的饥饿阶段（大型互联网企业的日均日志收集量均以亿为单位计量），反而面临海量日志的淹没风险。各类采集方案提供者所面临的主要挑战已不是日志采集技术本身，而是如何实现日志数据的结构化和规范化组织，实现更为高效的下游统计计算，提供符合业务特性的数据展现，以及为算法提供更便捷、灵活的支持等方面。

### 2.3.1 典型场景

这里介绍两个最典型的场景和阿里巴巴所采用的解决方案。

#### 1. 日志分流与定制处理

大型互联网网站的日志类型和日志规模都呈现出高速增长的态势，而且往往会出现短时间的流量热点爆发。这一特点，使得在日志服务器端采用集中统一的解析处理方案变得不可能，其要求在日志解析和处理过程中必须考虑业务分流（相互之间不应存在明显的影响，爆发热点不应干扰定常业务日志的处理）、日志优先级控制，以及根据业务特点实现定制处理。例如，对于电商网站而言，数据分析人员对位于点击流前

端的促销页面和位于后端的商品页面的关注点是不一样的,而这两类页面的流量又往往同等重要且庞大,如果采用统一的解析处理方案,则往往需要在资源浪费(尽可能多地进行预处理)和需求覆盖不全(仅对最重要的内容进行预处理)两个选择之间进行取舍。这种取舍的结果一般不是最优的。

考虑到阿里日志体量的规模和复杂度,分治策略从一开始便是阿里互联网日志采集体系的基本原则。这里以 PV 日志采集领域一个最浅显的例子来说明,与业界通用的第三方日志采集方案的日志请求路径几乎归一不同,阿里 PV 日志的请求位置(URL)是随着页面所在业务类型的不同而变化的。通过尽可能靠前地布置路由差异,就可以尽可能早地进行分流,降低日志处理过程中的分支判断消耗,并作为后续的计算资源调配的前提,提高资源利用效率。与业界方案的普遍情况相比,阿里的客户端日志采集代码的一个突出特点是实现了非常高的更新频次(业界大多以季度乃至年为单位更新代码,阿里则是以周/月为单位),并实现了更新的配置化。我们不仅考虑诸如日志分流处理之类的日志服务器端分布计算方案,而且将分类任务前置到客户端(从某种程度上讲,这才是真正的“分布式”!)以实现整个系统的效能最大化。最后可以在计算后端几乎无感知的情况下,承载更大的业务量并保证处理质量和效率。

## 2. 采集与计算一体化设计

以 PV 日志为例,页面 PV 日志采集之后一个基础性操作是日志的归类与汇总。在早期的互联网日志分析实践中,是以 URL 路径,继而以 URL(正则)规则集为依托来进行日志分类的。在网站规模较小时,这一策略还可以基本顺利地运转下去,但随着网站的大型化和开发人员的增加,URL 规则集的维护和使用成本会很快增长到不现实的程度,同时失控的大规模正则适配甚至会将日志计算硬件集群彻底榨干。

这一状况要求日志采集方案必须将采集与计算作为一个系统来考量,进行一体化设计。阿里日志采集针对这一问题给出的答案是两套日志规范和与之对应的元数据中心。其中,对应于 PV 日志的解决方案是目前用户可直观感知的 SPM 规范(例如,在页面的 URL 内可以看见 spm



参数)和SPM元数据中心。通过SPM的注册和简单部署(仅需要在页面文件内声明一个或多个标签),用户即可将任意的页面流量进行聚类,不需要进行任何多余的配置就可以在相应的内部数据产品内查询聚合统计得到的流量、转化漏斗、引导交易等数据,以及页面各元素点击数据的可视化视图。对应于自定义日志的解决方案则是黄金令箭(Goldlog)/APP端的点击或其他日志规范及其配置中心。通过注册一个与所在页面完全独立的令箭实体/控件实体,用户可以一键获得对应的埋点代码,并自动获得实时统计数据 and 与之对应的可视化视图。通过简单的扩展配置,用户还可以自动获得自定义统计维度下的分量数据。

在当前的互联网环境下,互联网日志的规模化采集方案必须具备一个与终端设备的技术特点无关,具有高度扩展弹性和适应性,同时深入契合应用需求的业务逻辑模型,并基于此制定对应的采集规范交由产品开发人员执行。若非如此,则不足以保障采集—解析—处理—应用整个流程的通畅。目前阿里已成功实现规范制定—元数据注册—日志采集—自动化计算—可视化展现全流程的贯通。通过一体化设计,用户甚至可以在不理解规范的前提下,通过操作向导式界面,实现日志采集规范的自动落地和统计应用。日志本身不是日志采集的目的,服务于基于日志的后续应用,才是日志采集正确的着眼点。

## 2.3.2 大促保障

日志数据在阿里系乃至整个电商系应该都是体量最大的一类数据,在“双11”期间,日志必然会暴涨,近万亿的数据量对日志全链路来说,无疑是不小的挑战(见图2.4)。

从端上埋点采集,到日志服务器收集,经过数据传输,再到日志实时解析、实时分析,任何一个环节出现问题,全链路保障就是失败的。考虑到服务器的收集能力(如峰值QPS等)、数据传输能力(TT的搬运速度)、实时解析的吞吐量、实时业务分析的处理能力,我们在各环节都做了不少的工作。

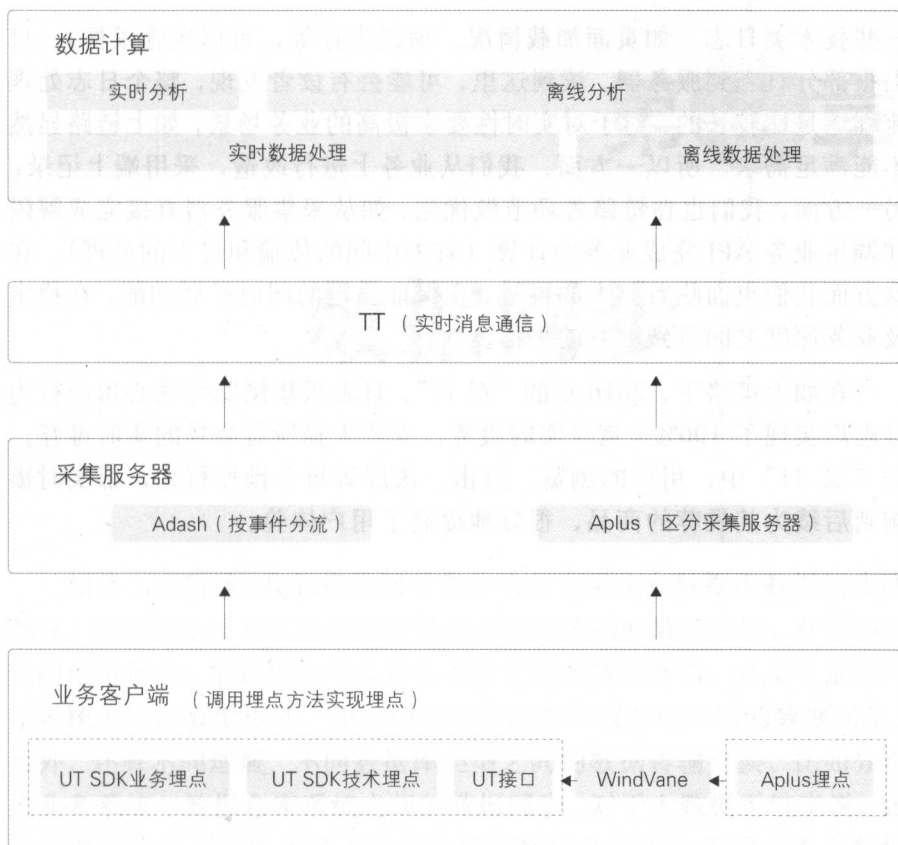


图 2.4 数据处理全链路

首先，端上实现了服务器端推送配置到客户端，且做到高到达率；其次，对日志做了分流，结合日志的重要程度及各类日志的大小，实现了日志服务器端的拆分；最后，在实时处理方面，也做了不少的优化以提高应用的吞吐量。在上面几项的基础上，结合实时处理能力，评估峰值数据量，在高峰期通过服务器端推送配置的方式对非重要日志进行适当限流，错峰后逐步恢复。此处说的服务器端推送配置包含较多的内容，首先是作用范围，可以针对应用、平台、事件、事件中的某个场景；其次是具体实施，包括延迟上报、部分采样等。所谓延迟上报，即配置生效后，满足条件的日志将被暂时存在客户端，待配置恢复后再上传到服务器；所谓采样，即配置生效后，满足条件的日志将被实施采样（对于

一些技术类日志，如页面加载情况、消耗内存等，可以实施采样)，只上报部分日志到服务器。读到这里，可能会有读者发现，整个日志处理流程还是比较长的，对于对实时性要求极高的业务场景，如上链路显然不能满足需求。所以一方面，我们从业务上进行改造，采用端上记录；另一方面，我们也在链路各环节做优化，如从采集服务器直接完成解码并调用业务 API 完成业务的计算（省去中间的传输和过多的处理）。在这方面我们也面临着巨大的挑战，在保证稳定的同时扩展功能，在稳定及业务深度之间做到很好的平衡。

在如上策略下，2016 年的“双 11”，日志采集浏览等核心用户行为日志均实现了 100% 全量及实时服务，支持天猫所有会场的实时推荐。在“双 11”中，用户的浏览、点击、滚屏等每个操作行为，都实时影响到后续为其推荐的商品，很好地提高了用户体验。

## 第3章

# 数据同步

如第1章所述,我们将数据采集分为日志采集和数据库数据同步两部分。数据同步技术更通用的含义是不同系统间的数据流转,有多种不同的应用场景。主数据库与备份数据库之间的数据备份,以及主系统与子系统之间的数据更新,属于同类型不同集群数据库之间的数据同步。另外,还有不同地域、不同数据库类型之间的数据传输交换,比如分布式业务系统与数据仓库系统之间的数据同步。对于大数据系统来说,包含数据从业务系统同步进入数据仓库和数据从数据仓库同步进入数据服务或数据应用两个方面。本章侧重讲解数据从业务系统同步进入数据仓库这个环节,但其适用性并不仅限于此。

### 3.1 数据同步基础

源业务系统的数据类型多种多样,有来源于关系型数据库的结构化数据,如 MySQL、Oracle、DB2、SQL Server 等;也有来源于非关系型数据库的非结构化数据,如 OceanBase、HBase、MongoDB 等,这类数

据通常存储在数据库表中；还有来源于文件系统的结构化或非结构化数据，如阿里云对象存储 OSS、文件存储 NAS 等，这类数据通常以文件形式进行存储。数据同步需要针对不同的数据类型及业务场景选择不同的同步方式。总的来说，同步方式可以分为三种：直连同步、数据文件同步和数据库日志解析同步。

### 3.1.1 直连同步

直连同步是指通过定义好的规范接口 API 和基于动态链接库的方式直接连接业务库，如 ODBC/JDBC 等规定了统一规范的标准接口，不同的数据库基于这套标准接口提供规范的驱动，支持完全相同的函数调用和 SQL 实现（见图 3.1）。

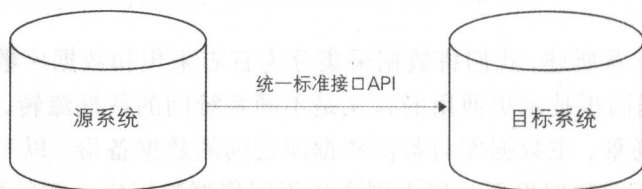


图 3.1 直连同步示意图

这种方式配置简单，实现容易，比较适合操作型业务系统的数据同步。但是业务库直连的方式对源系统的性能影响较大，当执行大批量数据同步时会降低甚至拖垮业务系统的性能。如果业务库采取主备策略，则可以从备库抽取数据，避免对业务系统产生性能影响。但是当数据量较大时，采取此种抽取方式性能较差，不太适合从业务系统到数据仓库系统的同步。

### 3.1.2 数据文件同步

数据文件同步通过约定好的文件编码、大小、格式等，直接从源系统生成数据的文本文件，由专门的文件服务器，如 FTP 服务器传输到目标系统后，加载到目标数据库系统中。当数据源包含多个异构的数据

库系统（如 MySQL、Oracle、SQL Server、DB2 等）时，用这种方式比较简单、实用。另外，互联网的日志类数据，通常是以文本文件形式存在的，也适合使用数据文件同步方式（见图 3.2）。

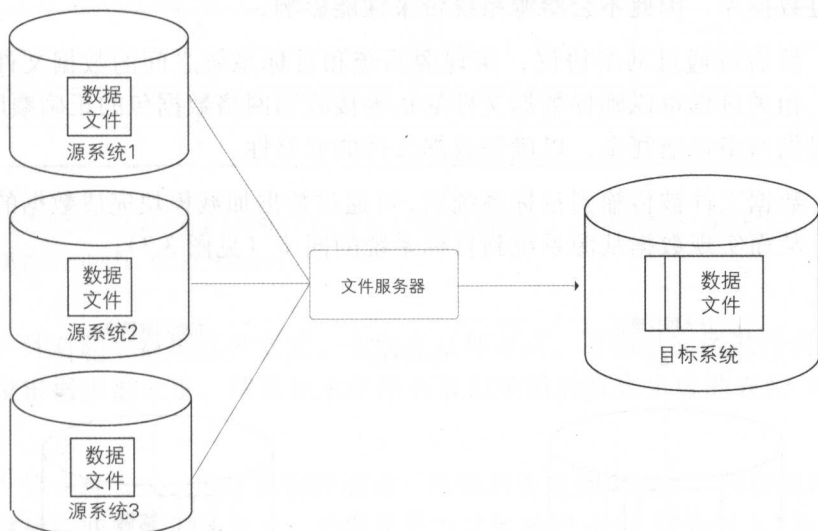


图 3.2 数据文件同步示意图

由于通过文件服务器上传、下载可能会造成丢包或错误，为了确保数据文件同步的完整性，通常除了上传数据文件本身以外，还会上传一个校验文件，该校验文件记录了数据文件的数据量以及文件大小等校验信息，以供下游目标系统验证数据同步的准确性。

另外，在从源系统生成数据文件的过程中，可以增加压缩和加密功能，传输到目标系统以后，再对数据进行解压缩和解密，这样可以大大提高文件的传输效率和安全性。

### 3.1.3 数据库日志解析同步

目前，大多数主流数据库都已经实现了使用日志文件进行系统恢复，因为日志文件信息足够丰富，而且数据格式也很稳定，完全可以通过解析日志文件获取发生变更的数据，从而满足增量数据同步的需求。

以 Oracle 为例，可以通过源系统的进程，读取归档日志文件用以收集变化的数据信息，并判断日志中的变更是否属于被收集对象，将其解析到目标数据文件中。这种读操作是在操作系统层面完成的，不需要通过数据库，因此不会给源系统带来性能影响。

然后可通过网络协议，实现源系统和目标系统之间的数据文件传输。相关进程可以确保数据文件的正确接收和网络数据包的正确顺序，并提供网络传输冗余，以确保数据文件的完整性。

数据文件被传输到目标系统后，可通过数据加载模块完成数据的导入，从而实现数据从源系统到目标系统的同步（见图 3.3）。

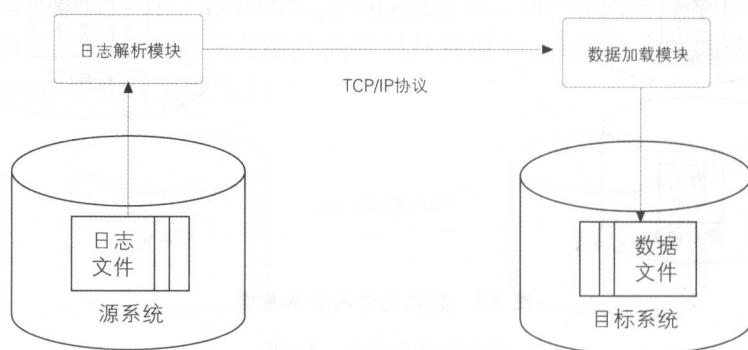


图 3.3 数据库日志解析同步示意图

数据库日志解析同步方式实现了实时与准实时同步的能力，延迟可以控制在毫秒级别，并且对业务系统的性能影响也比较小，目前广泛应用于从业务系统到数据仓库系统的增量数据同步应用之中。

由于数据库日志抽取一般是获取所有的数据记录的变更（增、删、改），落地到目标表时我们需要根据主键去重按照日志时间倒排序获取最后状态的变化情况。对于删除数据这种变更情况，针对不同的业务场景可以采用一些不同的落地手法。

我们以具体的实例进行说明。如表 3.1 所示为源业务系统中某表变更日志流水表。其含义是：存在 5 条变更日志，其中主键为 1 的记录有 3 条变更日志，主键为 2 的记录有 2 条变更日志。

表 3.1 某源表变更日志流水表

流水号	主键 (PK)	变更类型	数据内容
1	1	I	a=1, b=2
2	2	U	a=7, b=6
3	1	U	a=4, b=0
4	2	D	NULL
5	1	U	a=4, b=7

备注:

- (1) 变更类型中的 I 表示新增 (INSERT), U 表示更新 (UPDATE)、D 表示删除 (DELETE)。  
 (2) 数据内容中的 a、b 为此表的字段。

针对删除数据这种变更,主要有三种方式,下面以实例进行说明。假设根据主键去重,按照流水倒序获取记录最后状态生成的表为 delta 表。

第一种方式,不过滤删除流水。不管是否是删除操作,都获取同一主键最后变更的那条流水。采用此种方式生成的 delta 表如表 3.2 所示。

表 3.2 采用第一种方式生成的 delta 表

流水号	主键 (PK)	变更类型	字段 a	字段 b
4	2	D	NULL	NULL
5	1	U	4	7

第二种方式,过滤最后一条删除流水。如果同一主键最后变更的那条流水是

删除操作,就获取倒数第二条流水。采用此种方式生成的 delta 表如表 3.3 所示。

表 3.3 采用第二种方式生成的 delta 表

流水号	主键 (PK)	变更类型	字段 a	字段 b
2	2	U	7	6
5	1	U	4	7



第三种方式，过滤删除流水和之前的流水。如果在同一主键变更的过程中有删除操作，则根据操作时间将该删除操作对应的流水和之前的流水都过滤掉。采用此种方式生成的 delta 表如表 3.4 所示。

表 3.4 采用第三种方式生成的 delta 表

流水号	主键 (PK)	变更类型	字段 a	字段 b
5	1	U	4	7

对于采用哪种方式处理删除数据，要看前端是如何删除无效数据的。前端业务系统删除数据的方式一般有两种：正常业务数据删除和手工批量删除。手工批量删除通常针对类似的场景，业务系统只做逻辑删除，不做物理删除，DBA 定期将部分历史数据直接删除或者备份到备份库。

一般情况下，可以采用不过滤的方式来处理，下游通过是否删除记录的标识来判断记录是否有效。如果明确业务数据不存在业务上的删除，但是存在批量手工删除或备份数据删除，例如淘宝商品、会员等，则可以采用只过滤最后一条删除流水的方式，通过状态字段来标识删除记录是否有效。

通过数据库日志解析进行同步的方式性能好、效率高，对业务系统的影响较小。但是它也存在如下一些问题：

- 数据延迟。例如，业务系统做批量补录可能会使数据更新量超出系统处理峰值，导致数据延迟。
- 投入较大。采用数据库日志抽取的方式投入较大，需要在源数据库与目标数据库之间部署一个系统实时抽取数据。
- 数据漂移和遗漏。数据漂移，一般是对增量表而言的，通常是指该表的同一个业务日期数据中包含前一天或后一天凌晨附近的数据或者丢失当天的变更数据。这个问题我们将在“数据漂移的处理”一节中详细论述。

## 3.2 阿里数据仓库的同步方式

数据仓库的特性之一是集成，将不同的数据来源、不同形式的数据整合在一起，所以从不同业务系统将各类数据源同步到数据仓库是一切的开始。那么阿里数据仓库的数据同步有什么特别之处呢？

阿里数据仓库的数据同步的特点之一是数据来源的多样性。在传统的数据仓库系统中，一般数据来源于各种类型的关系型数据库系统，比如 MySQL、SQL Server、Oracle、DB2 等，这类数据的共同特点便是高度结构化，易于被计算机系统处理。而在大数据时代，除了结构化数据，还有大量非结构化数据，比如 Web 服务器产生的日志、各类图片、视频等。特别是日志数据，记录了用户对网站的访问情况，这类数据通常直接以文本文件形式记录在文件系统中，对于数据的分析、统计、挖掘等各类数据应用有极大的价值。以淘宝网为例，我们可以从用户浏览、点击页面的日志数据分析出用户的偏好和习惯，进而推荐适合的产品以提高成交率。

阿里数据仓库的数据同步的特点之二则体现在数据量上。传统的数据仓库系统每天同步的数据量一般在几百 GB 甚至更少，而一些大型互联网企业的大数据系统每天同步的数据量则达到 PB 级别。目前阿里巴巴的大数据处理系统 MaxCompute 的数据存储达到 EB 级别，每天需要同步的数据量达到 PB 级别，这种量级上的差距是巨大的。

数据源的类型是多样的，需要同步的数据是海量的，那该如何准确、高效地完成数据同步呢？这里就需要针对不同的数据源类型和数据应用的时效性要求而采取不同的策略。

### 3.2.1 批量数据同步

对于离线类型的数据仓库应用，需要将不同的数据源批量同步到数据仓库，以及将经过数据仓库处理的结果数据定时同步到业务系统。

当前市场上的数据库系统种类很多，有行存储的和列存储的，有开源的和非开源的，每一种数据库的数据类型都略有不同，而数据仓库系

统则是集成各类数据源的地方，所以数据类型是统一的。要实现各类数据库系统与数据仓库系统之间的批量双向数据同步，就需要先将数据转换为中间状态，统一数据格式。由于这类数据都是结构化的，且均支持标准的 SQL 语言查询，所以所有的数据类型都可以转换为字符串类型。因此，我们可以通过将各类源数据库系统的数据类型统一转换为字符串类型的方式，实现数据格式的统一。

阿里巴巴的 DataX 就是这样一个能满足多方向高自由度的异构数据交换服务产品。对于不同的数据源，DataX 通过插件的形式提供支持，将数据从数据源读出并转换为中间状态，同时维护好数据的传输、缓存等工作。数据在 DataX 中以中间状态存在，并在目标数据系统中将中间状态的数据转换为对应的数据格式后写入。目前 DataX 每天都需要处理 2PB 左右的批量数据同步任务，通过分布式模式，同步完所有的数据所需要的时间一般在 3 小时以内，有力保障了大数据同步的准确性及高效性（见图 3.4）。

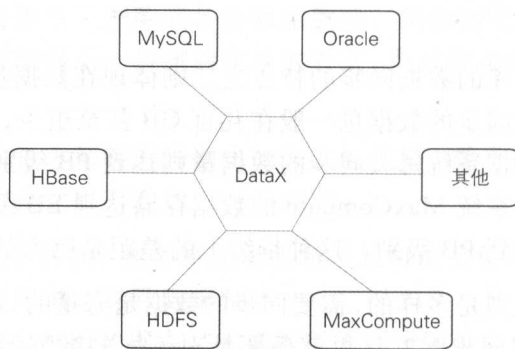


图 3.4 DataX 可接入的数据源

DataX 采用 Framework+Plugin 的开放式框架实现，Framework 处理缓冲、流程控制、并发、上下文加载等高速数据交换的大部分技术问题，并提供简单的接口与插件接入（见图 3.5）。插件仅需实现对数据处理系统的访问，编写方便，开发者可以在极短的时间内开发一个插件以快速支持新的数据库或文件系统。数据传输在单进程（单机模式）/多进程（分布式模式）下完成，传输过程全内存操作，不读写磁盘，也没有进

程间通信,实现了在异构数据库或文件系统之间的高速数据交换。

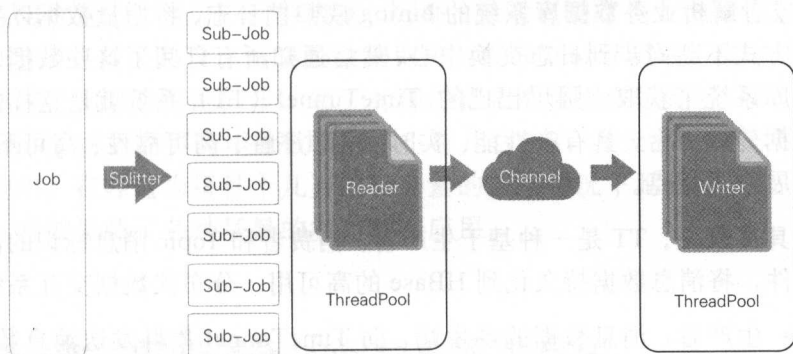


图 3.5 DataX 架构设计图

- Job: 数据同步作业。
- Splitter: 作业切分模块, 将一个大任务分解成多个可以并行的小任务。
- Sub-Job: 数据同步作业切分后的小任务, 或称之为 Task。
- Reader: 数据读入模块, 负责运行切分后的小任务, 将数据从源系统装载到 DataX。
- Channel: Reader 和 Writer 通过 Channel 交换数据。
- Writer: 数据写出模块, 负责将数据从 DataX 导入目标数据系统。

### 3.2.2 实时数据同步

对于日志类数据来说, 由于每天的日志是源源不断产生的, 并且分布在不同的服务器中, 有些大型互联网公司的服务器集群有成千上万台机器, 所以所产生的日志需要尽快以数据流的方式不间断地同步到数据仓库。另外, 还有一些数据应用, 需要对业务系统产生的数据进行实时处理, 比如天猫“双 11”的数据大屏, 对所产生的交易数据需要实时汇总, 实现秒级的数据刷新。这类数据是通过解析 MySQL 的 binlog 日志 (相当于 Oracle 的归档日志) 来实时获得增量的数据更新, 并通过消息订阅模式来实现数据的实时同步的。具体来说, 就是建立一个日志

数据交换中心，通过专门的模块从每台服务器源源不断地读取日志数据，或者解析业务数据库系统的 binlog 或归档日志，将增量数据以数据流的方式不断同步到日志交换中心，然后通知所有订阅了这些数据的数据仓库系统来获取。阿里巴巴的 TimeTunnel (TT) 系统就是这样的实时数据传输平台，具有高性能、实时性、顺序性、高可靠性、高可用性、可扩展性等特点。

具体来说，TT 是一种基于生产者、消费者和 Topic 消息标识的消息中间件，将消息数据持久化到 HBase 的高可用、分布式数据交互系统。

- 生产者：消息数据的产生端，向 TimeTunnel 集群发送消息数据，就是图 3.6 中的生产 Client。
- 消费者：消息数据的接收端，从 TimeTunnel 集群中获取数据进行业务处理。
- Topic：消息类型的标识，如淘宝 acookie 日志的 Topic 为 taobao\_acookie，生产 Client 和消费 Client 均需要知道对应的 Topic 名字。
- Broker 模块：负责处理客户端收发消息数据的请求，然后往 HBase 取发数据。

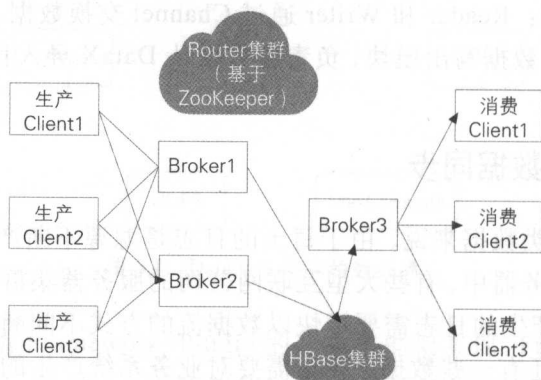


图 3.6 TimeTunnel 实时数据传输示意图

TimeTunnel 支持主动、被动等多种数据订阅机制，订阅端自动负载均衡，消费者自己把握消费策略。对于读写比例很高的 Topic，能够做

到读写分离,使消费不影响发送。同时支持订阅历史数据,可以随意设置订阅位置,方便用户回补数据。另外,针对订阅有强大的属性过滤功能,用户只需关心自己需要的数据即可。TimeTunnel 高效、稳定地支持阿里巴巴实时数据的同步,每天处理的日志类数据多达几百 TB,数据库 binlog 解析的实时增量数据同步也有几百 TB,在天猫“双 11”大促活动中,在峰值为每秒十几万笔交易量的极端情况下延迟控制在 3s 以内,有效保障了各种场景的实时数据应用。

### 3.3 数据同步遇到的问题与解决方案

在实际的大数据同步应用中会遇到各种各样的挑战,处理方法也是不断变化的,下面就针对常见的问题一起讨论一下解决方案。

#### 3.3.1 分库分表的处理

随着业务的不断增长,业务系统处理的数据量也在飞速增加,需要系统具备灵活的扩展能力和高并发大数据量的处理能力,目前一些主流数据库系统都提供了分布式分库分表方案来解决这个问题(见图 3.7)。但是对于数据同步来说,这种分库分表的设计无疑加大了同步处理的复杂度。试想一下,如果有一个中间表,具备将分布在不同数据库中的不同表集成为一个表的能力,就能让下游应用像访问单库单表一样方便。

阿里巴巴的 TDDL (Taobao Distributed Data Layer) 就是这样一个分布式数据库的访问引擎,通过建立中间状态的逻辑表来整合统一分库分表的访问(见图 3.8)。

TDDL 是在持久层框架之下、JDBC 驱动之上的中间件,它与 JDBC 规范保持一致,有效解决了分库分表的规则引擎问题,实现了 SQL 解析、规则计算、表名替换、选择执行单元并合并结果集的功能,同时解决了数据库表的读写分离、高性能主备切换的问题,实现了数据库配置信息的统一管理。

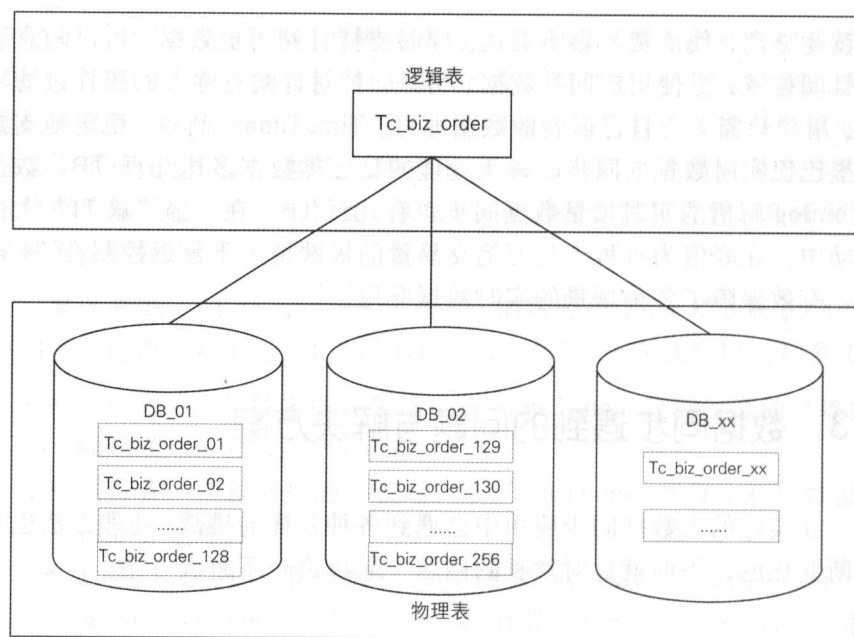


图 3.7 分库分表处理

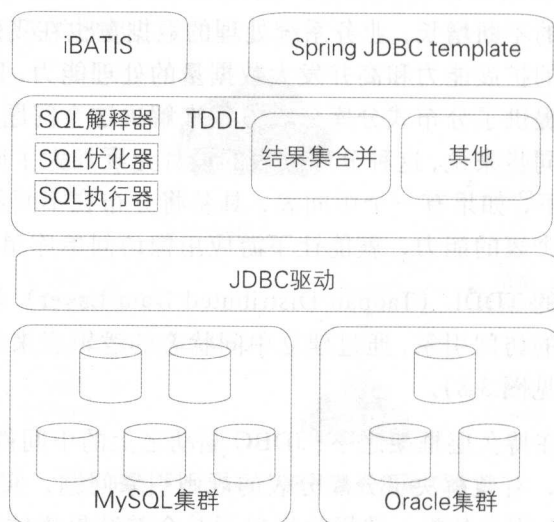


图 3.8 TDDL 分布式数据库访问引擎



### 3.3.2 高效同步和批量同步

数据同步的方法通常是先创建目标表,再通过同步工具的填写数据库连接、表、字段等各种配置信息后测试完成数据同步。这也是 DataX 任务的配置过程,同步中心对 DataX 进行进一步封装,通过源系统元数据降低了数据库连接、表和字段等信息的配置复杂度,但在实际生产过程中我们仍然会遇到一些问题。

- 随着业务的发展和变化,会新增大批量的数据同步,使用传统方式每天去完成成百上千的数据同步工作,一方面,工作量会特别大;另一方面,相似并且重复的操作会降低开发人员的工作热情。
- 数据仓库的数据源种类特别丰富,遇到不同类型的数据源同步就要求开发人员去了解其特殊配置。
- 部分真正的数据需求方,如 Java 开发和业务运营,由于存在相关数据同步的专业技能门槛,往往需要将需求提交给数据开发方来完成,额外增加了沟通和流程成本。

为了解决上述问题,阿里巴巴数据仓库研发了 OneClick 产品:

- 对不同数据源的数据同步配置透明化,可以通过库名和表名唯一定位,通过 IDB 接口获取元数据信息自动生成配置信息。
- 简化了数据同步的操作步骤,实现了与数据同步相关的建表、配置任务、发布、测试操作一键化处理,并且封装成 Web 接口进一步达到批量化的效果。
- 降低了数据同步的技能门槛,让数据需求方更加方便地获取和使用数据。

通过 OneClick 产品,真正实现了数据的一键化和批量化同步,一键完成 DDL 和 DML 的生成、数据的冒烟测试以及在生产环境中测试等。因此,阿里巴巴通过极少的人力投入,实现了数据同步的集中化建设和管理;改变了之前各数据开发人员自行同步带来的效率低、重复同步和同步配置质量低下等问题,大大降低了数据同步成本。

注:IDB 是阿里巴巴集团用于统一管理 MySQL、OceanBase、PostgreSQL、Oracle、SQL Server 等关系型数据库的平台,它是一种集数据管理、结构

管理、诊断优化、实时监控和系统管理于一体的数据管理服务；在对集团数据库表的统一管理服务过程中，IDB 产出了数据库、表、字段各个级别元数据信息，并且提供了元数据接口服务。

### 3.3.3 增量与全量同步的合并

在批量数据同步中，有些表的数据量随着业务的发展越来越大，如果按周期全量同步的方式会影响处理效率。在这种情况下，可以选择每次只同步新变更的增量数据，然后与上一个同步周期获得的全量数据进行合并，从而获得最新版本的全量数据。

在传统的数据整合方案中，合并技术大多采用 merge 方式 (update+insert)；当前流行的大数据平台基本都不支持 update 操作，现在我们比较推荐的方式是全外连接 (full outer join) + 数据全量覆盖重新加载 (insert overwrite)，即如日调度，则将当天的增量数据和前一天的全量数据做全外连接，重新加载最新的全量数据。在大数据量规模下，全量更新的性能比 update 要高得多。此外，如果担心数据更新错误问题，可以采用分区方式，每天保持一个最新的全量版本，保留较短的时间周期（如 3~7 天）。

另外，当业务系统的表有物理删除数据的操作，而数据仓库需要保留所有历史数据时，也可以选择这种方式，在数据仓库中永久保留最新的全量数据快照。下面我们以淘宝订单表的具体实例来说明。

淘宝交易订单表，每天新增、变更的增量数据多达几亿条，历史累计至今的全量数据则有几百亿条，面对如此庞大的数据量，如果每天从业务系统全量同步显然是不可能的。可行的方式是同步当天的增量数据，并与数据仓库中的前一天全量数据合并，获得截至当天的最新全量数据（见图 3.9）。

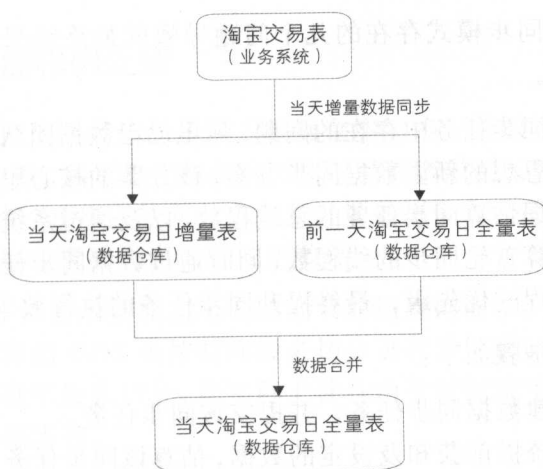


图 3.9 淘宝交易订单同步示意图

### 3.3.4 同步性能的处理

数据同步任务是针对不同数据库系统之间的数据同步问题而创建的一系列周期调度的任务。在大型的数据调度工作台上，每天会运行大量的数据同步任务。针对数据同步任务，一般首先需要设定首轮同步的线程数，然后运行同步任务。

这样的数据同步模式存在以下几个问题：

- 有些数据同步任务的总线程数达不到用户设置的首轮同步的线程数时，如果同步控制器将这些同步线程分发到 CPU 比较繁忙的机器上，将导致这些同步任务的平均同步速度非常低，数据同步速度非常慢。
- 用户不清楚该如何设置首轮同步的线程数，基本都会设置成一个固定的值，导致同步任务因得不到合理的 CPU 资源而影响同步效率。
- 不同的数据同步任务的重要程度是不一样的，但是同步控制器平等对待接收到的同步线程，导致重要的同步线程因得不到 CPU 资源而无法同步。

上述数据同步模式存在的几个问题导致的最终结果是数据同步任务运行不稳定。

针对数据同步任务中存在的问题,阿里巴巴数据团队实践出了一套基于负载均衡思想的新型数据同步方案。该方案的核心思想是通过目标数据库的元数据估算同步任务的总线程数,以及通过系统预先定义的期望同步速度估算首轮同步的线程数,同时通过数据同步任务的业务优先级决定同步线程的优先级,最终提升同步任务的执行效率和稳定性。

具体实现步骤如下:

- 用户创建数据同步任务,并提交该同步任务。
- 根据系统提前获知及设定的数据,估算该同步任务需要同步的数据量、平均同步速度、首轮运行期望的线程数、需要同步的总线程数。
- 根据需要同步的总线程数将待同步的数据拆分成相等数量的数据块,一个线程处理一个数据块,并将该任务对应的所有线程提交至同步控制器。
- 同步控制器判断需要同步的总线程数是否大于首轮运行期望的线程数,若大于,则跳转至;若不大于,则跳转至。
- 同步控制器采用多机多线程的数据同步模式,准备该任务第一轮线程的调度,优先发送等待时间最长、优先级最高且同一任务的线程。
- 同步控制器准备一定数据量(期望首轮线程数-总线程数)的虚拟线程,采用单机多线程的数据同步模式,准备该任务相应实体线程和虚拟线程的调度,优先发送等待时间最长、优先级最高且单机 CPU 剩余资源可以支持首轮所有线程数且同一任务的线程,如果没有满足条件的机器,则选择 CPU 剩余资源最多的机器进行首轮发送。
- 数据任务开始同步,并等待完成。
- 数据任务同步结束。

### 3.3.5 数据漂移的处理

通常我们把从源系统同步进入数据仓库的第一层数据称为 ODS 或者 staging 层数据, 阿里巴巴统称为 ODS。数据漂移是 ODS 数据的一个顽疾, 通常是指 ODS 表的同一个业务日期数据中包含前一天或后一天凌晨附近的数据或者丢失当天的变更数据。

由于 ODS 需要承接面向历史的细节数据查询需求, 这就需要物理落地到数据仓库的 ODS 表按时间段来切分进行分区存储, 通常的做法是按某些时间戳字段来切分, 而实际上往往由于时间戳字段的准确性问题导致发生数据漂移。

通常, 时间戳字段分为四类:

- 数据库表中用来标识数据记录更新时间的时间戳字段(假设这类字段叫 `modified_time`)。
- 数据库日志中用来标识数据记录更新时间的时间戳字段(假设这类字段叫 `log_time`)。
- 数据库表中用来记录具体业务过程发生时间的时间戳字段(假设这类字段叫 `proc_time`)。
- 标识数据记录被抽取到时间的时间戳字段(假设这类字段叫 `extract_time`)。

理论上, 这几个时间应该是一致的, 但是在实际生产中, 这几个时间往往会出现差异, 可能的原因有以下几点:

- 由于数据抽取是需要时间的, `extract_time` 往往会晚于前三个时间。
- 前台业务系统手工订正数据时未更新 `modified_time`。
- 由于网络或者系统压力问题, `log_time` 或者 `modified_time` 会晚于 `proc_time`。

通常的做法是根据其中的某一个字段来切分 ODS 表, 这就导致产生数据漂移。下面我们来具体看下数据漂移的几种场景。

- 根据 `extract_time` 来获取数据。这种情况数据漂移的问题最明显。
- 根据 `modified_time` 限制。在实际生产中这种情况最常见, 但是往往会发生不更新 `modified_time` 而导致的数据遗漏, 或者凌晨

时间产生的数据记录漂移到后一天。

- 根据 `log_time` 限制。由于网络或者系统压力问题，`log_time` 会晚于 `proc_time`，从而导致凌晨时间产生的数据记录漂移到后一天。例如，在淘宝“双 11”大促期间凌晨时间产生的数据量非常大，用户支付需要调用多个接口，从而导致 `log_time` 晚于实际的支付时间。
- 根据 `proc_time` 限制。仅仅根据 `proc_time` 限制，我们所获取的 ODS 表只是包含一个业务过程所产生的记录，会遗漏很多其他过程的变化记录，这违背了 ODS 和业务系统保持一致的设计原则。

处理方法主要有以下两种：

#### (1) 多获取后一天的数据

既然很难解决数据漂移的问题，那么就在 ODS 每个时间分区中向前、向后多冗余一些数据，保障数据只会多不会少，而具体的数据切分让下游根据自身不同的业务场景用不同的业务时间 `proc_time` 来限制。但是这种方式会有一些数据误差，例如一个订单是当天支付的，但是第二天凌晨申请退款关闭了该订单，那么这条记录的订单状态会被更新，下游在统计支付订单状态时会出现错误。

#### (2) 通过多个时间戳字段限制时间来获取相对准确的数据

- 首先根据 `log_time` 分别冗余前一天最后 15 分钟的数据和后一天凌晨开始 15 分钟的数据，并用 `modified_time` 过滤非当天数据，确保数据不会因为系统问题而遗漏。
- 然后根据 `log_time` 获取后一天 15 分钟的数据；针对此数据，按照主键根据 `log_time` 做升序排列去重。因为我们需要获取的是最接近当天记录变化的数据（数据库日志将保留所有变化的数据，但是落地到 ODS 表的是根据主键去重获取最后状态变化的数据）。
- 最后将前两步的结果数据做全外连接，通过限制业务时间 `proc_time` 来获取我们所需要的数据。

下面来看处理淘宝交易订单的数据漂移的实际案例。

我们在处理“双 11”交易订单时发现，有一大批在 11 月 11 日

23:59:59 左右支付的交易订单漂移到了 12 日。主要原因是用户下单支付后系统需要调用支付宝的接口而有所延迟,从而导致这些订单最终生成的时间跨天了。即 `modified_time` 和 `log_time` 都晚于 `proc_time`。

如果订单只有一个支付业务过程,则可以用支付时间来限制就能获取到正确的数据。但是往往实际订单有多个业务过程:下单、支付、成功,每个业务过程都有相应的时间戳字段,并不只有支付数据会漂移。

如果直接通过多获取后一天的数据,然后限制这些时间,则可以获取到相关数据,但是后一天的数据可能已经更新多次,我们直接获取到的那条记录已经是更新多次后的状态,数据的准确性存在一定的问题。

因此,我们可以根据实际情况获取后一天 15 分钟的数据,并限制多个业务过程的时间戳字段(下单、支付、成功)都是“双 11”当天的,然后对这些数据按照订单的 `modified_time` 做升序排列,获取每个订单首次数据变更的那条记录。

此外,我们可以根据 `log_time` 分别冗余前一天最后 15 分钟的数据和后一天凌晨开始 15 分钟的数据,并用 `modified_time` 过滤非当天数据,针对每个订单按照 `log_time` 进行降序排列,取每个订单当天最后一次数据变更的那条记录。

最后将两份数据根据订单做全外连接,将漂移数据回补到当天数据中。



## 第4章

# 离线数据开发

从采集系统中收集了大量的原始数据后，数据只有被整合和计算，才能被用于洞察商业规律，挖掘潜在信息，从而实现大数据价值，达到赋能于商业和创造价值的目的。面对海量的数据和复杂的计算，阿里巴巴的数据计算层包括两大体系：数据存储及计算平台（离线计算平台 MaxCompute 和实时计算平台 StreamCompute）、数据整合及管理体系（OneData）。

本章主要介绍 MaxCompute 和阿里巴巴内部基于 MaxCompute 的大数据开发套件，并对在数据开发过程中经常遇到的问题和相关解决方案进行介绍。

### 4.1 数据开发平台

阿里数据研发岗位的工作大致可以概括为：了解需求→模型设计→ETL 开发→测试→发布上线→日常运维→任务下线。与传统的数据仓库

开发（ETL）相比，阿里数据研发有如下几个特点：

- 业务变更频繁——业务发展非常快，业务需求多且变更频繁。
- 需要快速交付——业务驱动，需要快速给出结果。
- 频繁发布上线——迭代周期以天为单位，每天需要发布数次。
- 运维任务多——在集团公共层平均每个开发人员负责 500 多个任务。
- 系统环境复杂——阿里平台系统多为自研，且为了保证业务的发展，平台系统的迭代速度较快，平台的稳定性压力较大。

通过统一的计算平台（MaxCompute）、统一的开发平台（D2 等相关平台和工具）、统一的数据模型规范和统一的数据研发规范，可以在一定程度上解决数据研发的痛点。

### 4.1.1 统一计算平台

阿里离线数据仓库的存储和计算都是在阿里云大数据计算服务 MaxCompute 上完成的。

大数据计算服务 MaxCompute 是由阿里云自主研发的海量数据处理平台，主要服务于海量数据的存储和计算，提供完善的数据导入方案，以及多种经典的分布式计算模型，提供海量数据仓库的解决方案，能够更快速地解决用户的海量数据计算问题，有效降低企业成本，并保障数据安全。

MaxCompute 采用抽象的作业处理框架，将不同场景的各种计算任务统一在同一个平台之上，共享安全、存储、数据管理和资源调度，为来自不同用户需求的各种数据处理任务提供统一的编程接口和界面。它提供数据上传/下载通道、SQL、MapReduce、机器学习算法、图编程模型和流式计算模型多种计算分析服务，并且提供完善的安全解决方案。

#### 1. MaxCompute 的体系架构

MaxCompute 的体系架构如图 4.1 所示。

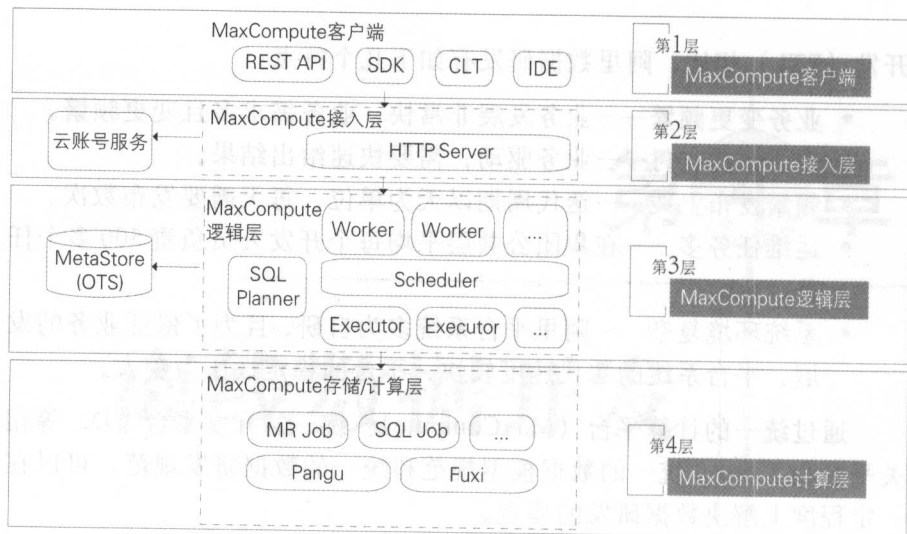


图 4.1 MaxCompute 体系架构图

MaxCompute 由四部分组成，分别是客户端（MaxCompute Client）、接入层（MaxCompute Front End）、逻辑层（MaxCompute Server）及存储与计算层（Apsara Core）。

MaxCompute 客户端有以下几种形式。

- Web：以 RESTful API 的方式提供离线数据处理服务。
- SDK：对 RESTful API 的封装，目前有 Java 等版本的实现。
- CLT（Command Line Tool）：运行在 Windows/Linux 下的客户端工具，通过 CLT 可以提交命令完成 Project 管理、DDL、DML 等操作。
- IDE：上层可视化 ETL/BI 工具，即阿里内部名称是在云端（D2），用户可以基于在云端完成数据同步、任务调度、报表生成等常见操作。

接入层提供 HTTP 服务、Cache、负载均衡，实现用户认证和服务层面的访问控制。

逻辑层又称作控制层，是 MaxCompute 的核心部分，实现用户空间和对象的管理、命令的解析与执行逻辑、数据对象的访问控制与授权等功能。在逻辑层有 Worker、Scheduler 和 Executor 三个角色：

- Worker 处理所有的 RESTful 请求，包括用户空间（Project）管理操作、资源（Resource）管理操作、作业管理等，对于 SQL DML、MR 等需要启动 MapReduce 的作业，会生成 MaxCompute Instance（类似于 Hive 中的 Job），提交给 Scheduler 进一步处理。
- Scheduler 负责 MaxCompute Instance 的调度和拆解，并向计算层的计算集群询问资源占用情况以进行流控。
- Executor 负责 MaxCompute Instance 的执行，向计算层的计算集群提交真正的计算任务。

计算层就是飞天内核（Apsara Core），运行在和控制层相互独立的计算集群上，它包括 Pangu（分布式文件系统）、Fuxi（资源调度系统）、Nuwa/ZK（Namespace 服务）、Shennong（监控模块）等。MaxCompute 中的元数据存储在美国云计算的另一个开放服务 OTS（Open Table Service，开放结构化数据服务）中，元数据内容主要包括用户空间元数据、Table/Partition Schema、ACL、Job 元数据、安全体系等。

## 2. MaxCompute 的特点

### （1）计算性能高且更加普惠

2016 年 11 月 10 日，Sort Benchmark 在官方网站公布了 2016 年排序竞赛 CloudSort 项目的最终成绩。阿里云以 \$1.44/TB 的成绩获得 Indy（专用目的排序）和 Daytona（通用目的排序）两个子项的世界冠军，打破了 AWS 在 2014 年保持的纪录 \$4.51/TB。这意味着阿里云将世界顶级的计算能力，变成普惠科技的云产品。CloudSort 又被称为“云计算效率之争”，这项目赛比拼的是完成 100TB 数据排序谁的花费更少，也是 Sort Benchmark 的各项比赛当中最具现实意义的项目之一。

## (2) 集群规模大且稳定性高

MaxCompute 平台共有几万台机器、存储近 1000PB，支撑着阿里巴巴的很多业务系统，包括数据仓库、BI 分析和决策支持、信用评估和无担保贷款风险控制、广告业务、每天几十亿流量的搜索和推荐相关性分析等，系统运行非常稳定。同时，MaxCompute 能保证数据的正确性，如对数据的准确性要求非常高的蚂蚁金服小额贷款业务，就运行于 MaxCompute 平台之上。

## (3) 功能组件非常强大

- MaxCompute SQL：标准 SQL 的语法，提供各类操作和函数来处理数据。
- MaxCompute MapReduce：提供 Java MapReduce 编程模型，通过接口编写 MR 程序处理 MaxCompute 中的数据。还提供基于 MapReduce 的扩展模型 MR2，在该模型下，一个 Map 函数后可以接入连续多个 Reduce 函数，执行效率比普通的 MapReduce 模型高。
- MaxCompute Graph：面向迭代的图计算处理框架，典型应用有 PageRank、单源最短距离算法、K-均值聚类算法。
- Spark：使用 Spark 接口编程处理存储在 MaxCompute 中的数据。
- RMaxCompute：使用 R 处理 MaxCompute 中的数据。
- Volume：MaxCompute 以 Volume 的形式支持文件，管理非二维表数据。

## (4) 安全性高

MaxCompute 提供功能强大的安全服务，为用户的数据安全提供保护。MaxCompute 采用多租户数据安全体系，实现用户认证、项目空间的用户与授权管理、跨项目空间的资源分享，以及项目空间的数据保护。如支付宝数据，符合银行监管的安全性要求，支持各种授权鉴权审查和“最小访问权限”原则，确保数据安全。

### 4.1.2 统一开发平台

阿里数据开发平台集成了多个子系统来解决实际生产中的各种痛点。围绕 MaxCompute 计算平台，从任务开发、调试、测试、发布、监控、报警到运维管理，形成了整套工具和产品，既提高了开发效率，又保证了数据质量，并且在确保数据产出时效的同时，能对数据进行有效管理。

数据研发人员完成需求了解和模型设计之后，进入开发环节，开发 workflow 如图 4.2 所示。

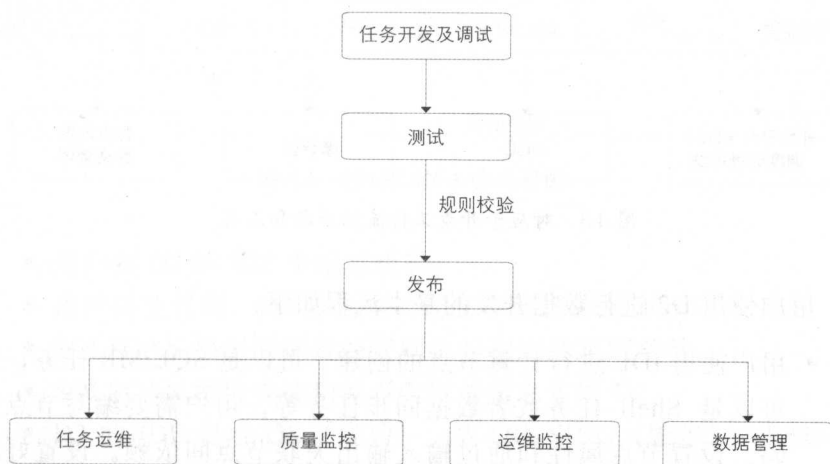


图 4.2 开发工作流图

对应于开发工作流的产品和工具如图 4.3 所示，我们将对其功能进行简要介绍。

#### 1. 在云端 (D2)

D2 是集成任务开发、调试及发布，生产任务调度及大数据运维，数据权限申请及管理等功能的一站式数据开发平台，并能承担数据分析工作台的功能。

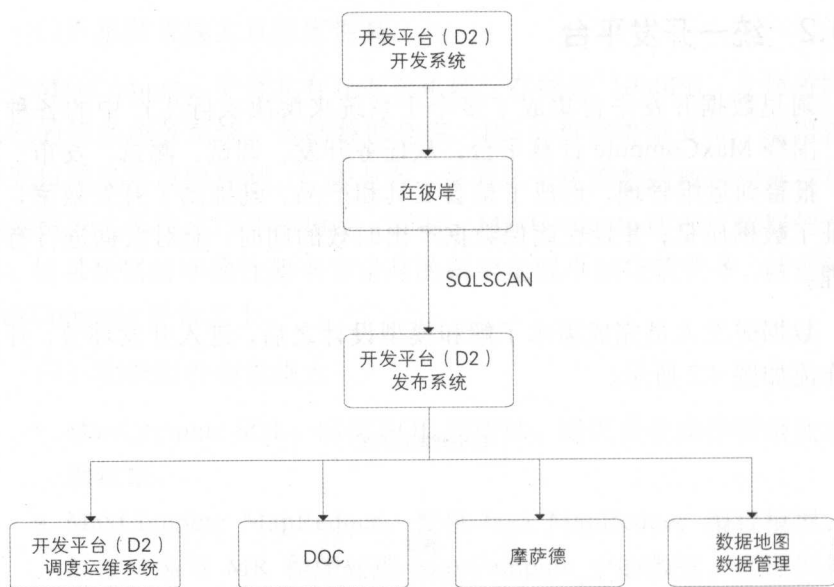


图 4.3 对应于开发工作流的产品和工具

用户使用 D2 进行数据开发的基本流程如下：

- 用户使用 IDE 进行计算节点的创建，可以是 SQL/MR 任务，也可以是 Shell 任务或者数据同步任务等，用户需要编写节点代码、设置节点属性和通过输入输出关联节点间依赖。设置好这些后，可以通过试运行来测试计算逻辑是否正确、结果是否符合预期。
- 用户点击提交，节点进入开发环境中，并成为某个工作流的其中一个节点。整个工作流可以被触发调度，这种触发可以是人为的（称之为“临时工作流”），也可以是系统自动的（称之为“日常工作流”）。当某个节点满足所有触发条件后，会被下发到调度系统的执行引擎 Alisa 中，完成资源分配和执行的整个过程。
- 如果节点在开发环境中运行无误，用户可以点击发布，将该节点正式提交到生产环境中，成为线上生产链路的一个环节。



## 2. SQLSCAN

SQLSCAN 将在任务开发中遇到的各种问题，如用户编写的 SQL 质量差、性能低、不遵守规范等，总结后形成规则，并通过系统及研发流程保障，事前解决故障隐患，避免事后处理。

SQLSCAN 与 D2 进行结合，嵌入到开发流程中，用户在提交代码时会触发 SQLSCAN 检查。SQLSCAN 工作流程如图 4.4 所示。



图 4.4 SQLSCAN 工作流程图

- 用户在 D2 的 IDE 中编写代码。
- 用户提交代码，D2 将代码、调度等信息传到 SQLSCAN。
- SQLSCAN 根据所配置的规则执行相应的规则校验。
- SQLSCAN 将检查成功或者失败的信息传回 D2。
- D2 的 IDE 显示 OK（成功）、WARNING（警告）、FAILED（失败，禁止用户提交）等消息。

SQLSCAN 主要有如下三类规则校验：

- 代码规范类规则，如表命名规范、生命周期设置、表注释等。
- 代码质量类规则，如调度参数使用检查、分母为 0 提醒、NULL 值参与计算影响结果提醒、插入字段顺序错误等。
- 代码性能类规则，如分区裁剪失效、扫描大表提醒、重复计算检测等。

SQLSCAN 规则有强规则和弱规则两类。触发强规则后，任务的提交会被阻断，必须修复代码后才能再次提交；而触发弱规则，则只会显示违反规则的提示，用户可以继续提交任务。

### 3. DQC

DQC (Data Quality Center, 数据质量中心) 主要关注数据质量, 通过配置数据质量校验规则, 自动在数据处理任务过程中进行数据质量方面的监控。

DQC 主要有数据监控和数据清洗两大功能。数据监控, 顾名思义, 能监控数据质量并报警, 其本身不对数据产出进行处理, 需要报警接收人判断并决定如何处理; 而数据清洗则是将不符合既定规则的数据清洗掉, 以保证最终数据产出不含“脏数据”, 数据清洗不会触发报警。

DQC 数据监控规则有强规则和弱规则之分, 强规则会阻断任务的执行 (将任务置为失败状态, 其下游任务将不会被执行); 而弱规则只告警而不会阻断任务的执行。常见的 DQC 监控规则有: 主键监控、表数据量及波动监控、重要字段的非空监控、重要枚举字段的离散值监控、指标值波动监控、业务规则监控等。

阿里数据仓库的数据清洗采用非侵入式的清洗策略, 在数据同步过程中不进行数据清洗, 避免影响数据同步的效率, 其过程在数据进入 ODS 层之后执行。对于需要清洗的表, 首先在 DQC 配置清洗规则; 对于离线任务, 每隔固定的时间间隔, 数据入仓之后, 启动清洗任务, 调用 DQC 配置的清洗规则, 将符合清洗规则的数据清洗掉, 并保存至 DIRTY 表归档。如果清洗掉的数据量大于预设的阈值, 则阻断任务的执行; 否则不会阻断。

DQC 工作流程如图 4.5 所示。

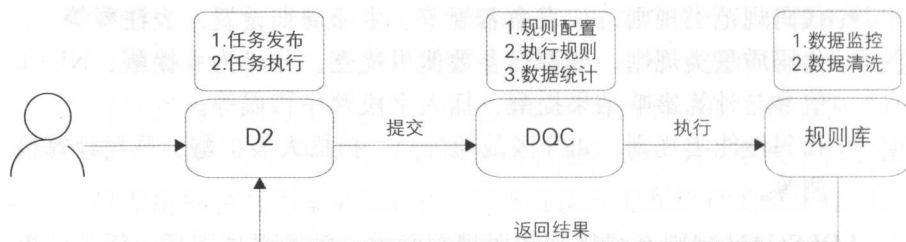


图 4.5 DQC 工作流程图

#### 4. 在彼岸

数据测试的典型测试方法是功能测试,主要验证目标数据是否符合预期。其主要有如下场景:

##### (1) 新增业务需求

新增产品经理、运营、BI 等的报表、应用或产品需求,需要开发新的 ETL 任务,此时应对上线前的 ETL 任务进行测试,确保目标数据符合业务预期,避免业务方根据错误数据做出决策。其主要对目标数据和源数据进行对比,包括数据量、主键、字段空值、字段枚举值、复杂逻辑(如 UDF、多路分支)等的测试。

##### (2) 数据迁移、重构和修改

由于数据仓库系统迁移、源系统业务变化、业务需求变更或重构等,需要对现有的代码逻辑进行修改,为保证数据质量需要对修改前后的数据进行对比,包括数据量差异、字段值差异对比等,保证逻辑变更正确。为了严格保证数据质量,对于优先级(优先级的定义见“数据质量”章节)大于某个阈值的任务,强制要求必须使用在彼岸进行回归测试,在彼岸回归测试通过之后,才允许进入发布流程。

在彼岸则是用于解决上述测试问题而开发的大数据系统的自动化测试平台,将通用的、重复性的操作沉淀在测试平台中,避免被“人肉”,提高测试效率。

在彼岸主要包含如下组件,除满足数据测试的数据对比组件之外,还有数据分布和数据脱敏组件。

- 数据对比:支持不同集群、异构数据库的表做数据对比。表级对比规则主要包括数据量和全文对比;字段级对比规则主要包括字段的统计值(如 SUM、AVG、MAX、MIN 等)、枚举值、空值、去重数、长度值等。
- 数据分布:提取表和字段的一些特征值,并将这些特征值与预期值进行比对。表级数据特征提取主要包括数据量、主键等;字段

级数据特征提取主要包括字段枚举值分布、空值分布、统计值（如 SUM、AVG、MAX、MIN 等）、去重数、长度值等。

- 数据脱敏：将敏感数据模糊化。在数据安全的大前提下，实现线上数据脱敏，在保证数据安全的同时又保持数据形态的分布，以便业务联调、数据调研和数据交换。

使用在彼岸进行回归测试的流程如图 4.6 所示。

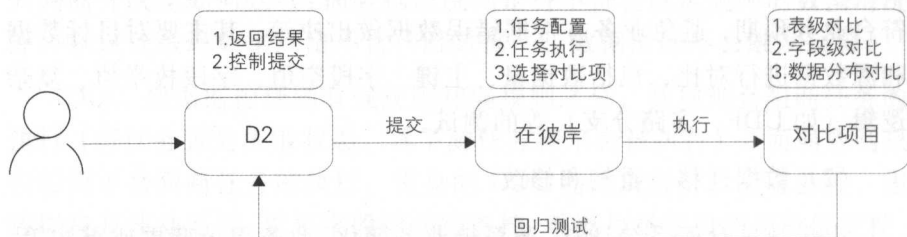


图 4.6 使用在彼岸进行回归测试流程图

## 4.2 任务调度系统

### 4.2.1 背景

现代信息化条件下的战争，从太空的卫星到空中的各类作战飞机，从地面的导弹到坦克火炮，从水面的大小舰艇到水下的潜艇，还有诸如网络、电磁环境等多种方式、多种维度的作战空间，各种武器装备、人员、作战环境纷繁复杂，如何能够准确、合理地调配这些资源，组织有序、高效的攻防体系赢得胜利，最关键的是需要有一个强大的指挥系统。在云计算大数据时代，调度系统无疑是整个大数据体系的指挥中枢。

如图 4.7 所示，调度系统中的各类任务互相依赖，形成一个典型的有向无环图。在传统的数据仓库系统中，很多是依靠 Crontab 定时任务功能进行任务调度处理的。这种方式有很多弊端：①各任务之间的依赖基于执行时间实现，容易造成前面的任务未结束或失败而后面的任务已

经运行；②任务难以并发执行，增加了整体的处理时间；③无法设置任务优先级；④任务的管理维护很不方便，无法进行执行效果分析等。

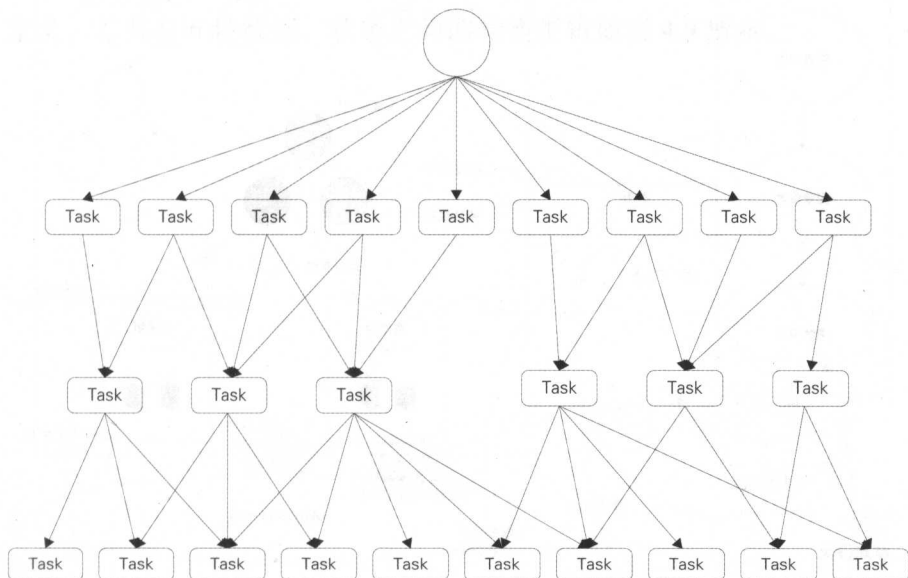


图 4.7 调度任务关系依赖图

而在大数据环境下，每天需要处理海量的任务，多的可以达到几十上百万。另外，任务的类型也很繁杂，有 MapReduce、Hive、SQL、Spark、Java、Shell、Python、Perl、虚拟节点等，任务之间互相依赖且需要不同的运行环境。为了解决以上问题，阿里巴巴的大数据调度系统应运而生。

## 4.2.2 介绍

### 1. 数据开发流程与调度系统的关系

数据开发流程与调度系统的关系如图 4.8 所示。

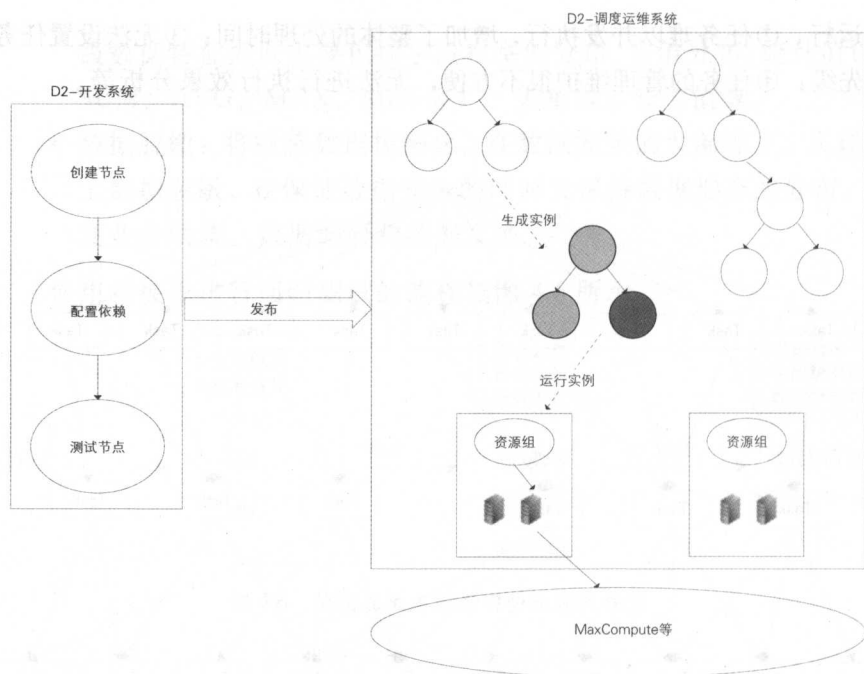


图 4.8 数据开发流程与调度系统的关系图

用户通过 D2 平台提交、发布的任务节点，需要通过调度系统，按照任务的运行顺序调度运行。

## 2. 调度系统的核心设计模型

整个调度系统共有两个核心模块：调度引擎（Phoenix Engine）和执行引擎（Alisa）。简单来说，调度引擎的作用是根据任务节点属性以及依赖关系进行实例化，生成各类参数的实值，并生成调度树；执行引擎的作用是根据调度引擎生成的具体任务实例和配置信息，分配 CPU、内存、运行节点等资源，在任务对应的执行环境中运行节点代码。

在介绍调度引擎设计之前，我们先来了解两个模型：任务状态机模型和工作流状态机模型。

### 3. 任务状态机模型

任务状态机模型是针对数据任务节点在整个运行生命周期的状态定义，总共有 6 种状态，状态之间的转换逻辑如图 4.9 所示。

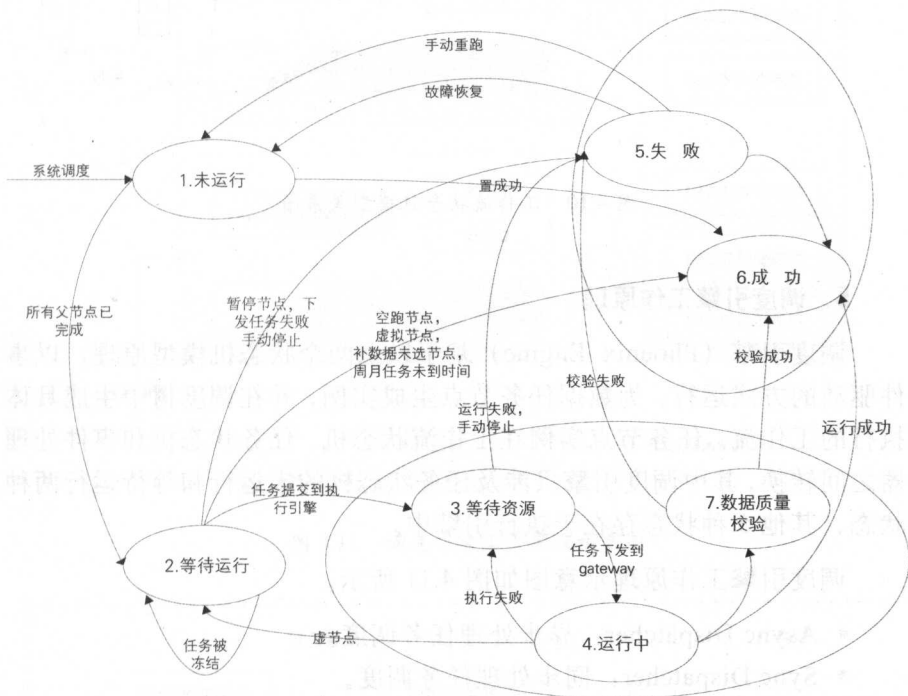


图 4.9 任务状态机模型关系图

### 4. workflow 状态机模型

workflow 状态机模型是针对数据任务节点在调度树中生成的 workflow 运行的不同状态定义，共有 5 种状态，其关系如图 4.10 所示。



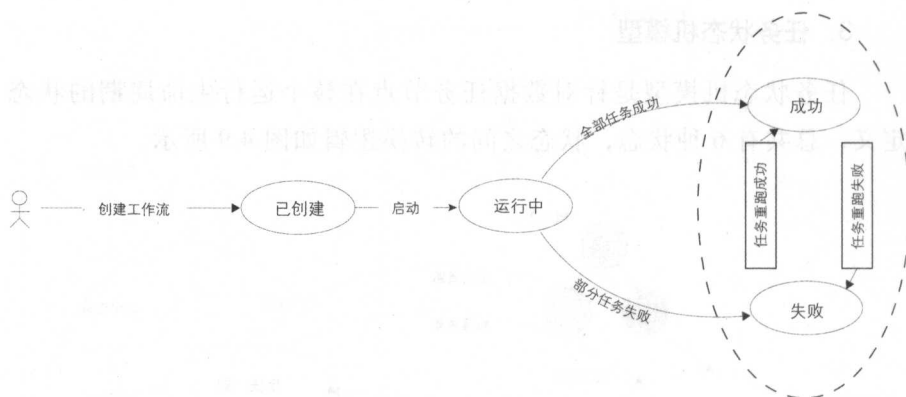


图 4.10 工作流状态机模型关系图

## 5. 调度引擎工作原理

调度引擎（Phoenix Engine）基于以上两个状态机模型原理，以事件驱动的方式运行，为数据任务节点生成实例，并在调度树中生成具体执行的工作流。任务节点实例在工作流状态机、任务状态机和事件处理器之间转换，其中调度引擎只涉及任务状态机的未运行和等待运行两种状态，其他 5 种状态存在于执行引擎中。

调度引擎工作原理示意图如图 4.11 所示。

- Async Dispatcher：异步处理任务调度。
  - Sync Dispatcher：同步处理任务调度。
  - Task 事件处理器：任务事件处理器，与任务状态机交互。
  - DAG 事件处理器：工作流事件处理器，与工作流状态机交互。
- 一个 DAG 事件处理器包含若干个 Task 事件处理器。

## 6. 执行引擎工作原理

首先来看看执行引擎（Alisa）的逻辑架构，如图 4.12 所示。

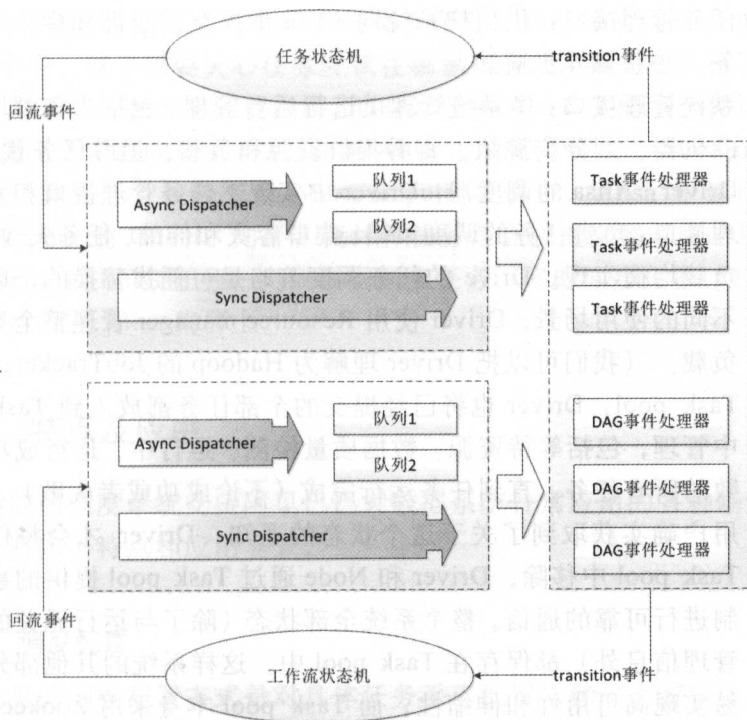


图 4.11 调度引擎工作原理示意图

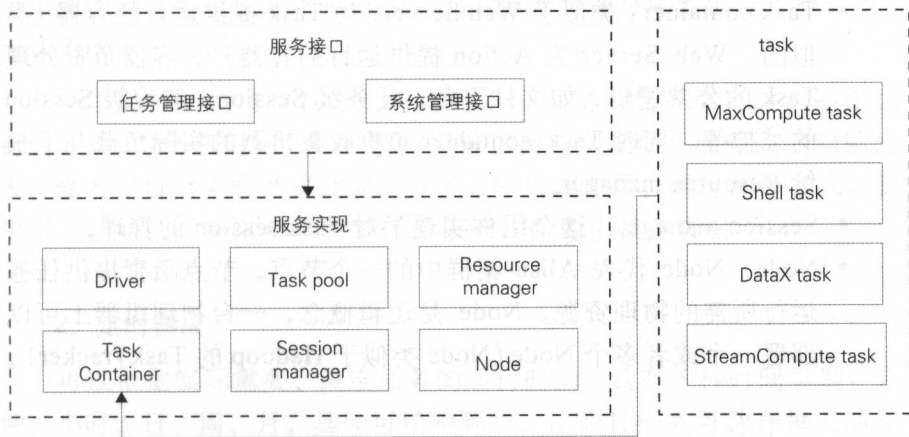


图 4.12 执行引擎逻辑架构图

- 任务管理接口：供用户系统向 Alisa 中提交、查询和操作离线任务，并获得异步通知。
- 系统管理接口：供系统管理员进行后台管理，包括为集群增加新的机器、划分资源组、查看集群资源和负载、追踪任务状态等。
- Driver：Alisa 的调度器，Driver 中实现了任务管理接口和系统管理接口；负责任务的调度策略、集群容灾和伸缩、任务失效备援、负载均衡实现。Driver 的任务调度策略是可插拔替换的，以满足不同的使用场景。Driver 使用 Resource manager 管理整个集群的负载。（我们可以把 Driver 理解为 Hadoop 的 JobTracker。）
- Task pool：Driver 也将已经提交的全部任务都放入到 Task pool 中管理，包括等待资源、数据质量检测、运行中、运行成功和失败的所有任务。直到任务运行完成（不论成功或者失败），并且用户确实获取到了关于这个状态的通知，Driver 才会将任务从 Task pool 中移除。Driver 和 Node 通过 Task pool 提供的事件机制进行可靠的通信。整个系统全部状态（除了与运行无关的部分管理信息外）都保存在 Task pool 中，这样系统的其他部分很容易实现高可用性和伸缩性。而 Task pool 本身采用 Zookeeper 实现，这样它本身也是具备高可用能力的。
- Resource manager：这个组件专注于集群整体资源的管理。
- Task container：类似于 Web Server，为 Task 提供运行的容器（类似的，Web Server 为 Action 提供运行的容器）。容器负责处理 Task 的公共逻辑，如文件下载，任务级 Session、流程级 Session 的维护等。同时 Task container 负责收集机器的实际负载并上报给 Resource manager。
- Session manager：这个组件实现了对 Task session 的管理。
- Node：Node 代表 Alisa 集群中的一个节点。节点负责提供任务运行所需的物理资源。Node 是逻辑概念，一台物理机器上可以部署一个或者多个 Node（Node 类似于 Hadoop 的 TaskTracker）。

## 7. 执行引擎的用法

Alisa 的用户系统包括上文的工作流服务、数据同步服务，以及调

度引擎生成的各类数据处理任务的调度服务。这样系统将任务提交到 Alisa 中后, 就不需要关心任务应该在哪里执行以及如何被执行了, 于是大大降低了系统本身的复杂度。同时其任务可以共享同一个物理集群资源, 提高了资源的利用效率。如果 Alisa 中的任务是一个 MaxCompute 任务, 计算实际会被提交到 MaxCompute 集群中, Alisa 中仅仅运行 MaxCompute 的 Client; 同样, 流计算任务等会被提交到对应的目标系统中运行; 而 Shell 任务、离线数据同步任务、实时同步任务 (TT) 等将直接运行在 Alisa 集群上。

### 4.2.3 特点及应用

当前的调度系统支持阿里巴巴大数据系统日常应用的各种场景, 其主要具有如下特点和应用场景。

#### 1. 调度配置

常见的调度配置方式是对具体任务手工配置依赖的上游任务, 此方式基本可以满足调度系统的正常运行。这种方式存在两个问题: 一是配置上较麻烦, 需要知道上游依赖表的产出任务; 二是上游任务修改不再产出依赖表或本身任务不再依赖某上游任务时, 对调度依赖不做修改, 导致依赖配置错误。

阿里巴巴的调度配置方式采用的是输入输出配置和自动识别相结合的方式。任务提交时, SQL 解析引擎自动识别此任务的输入表和输出表, 输入表自动关联产出此表的任务, 输出表亦然。通过此种方式, 解决了上述问题, 可以自动调整任务依赖, 避免依赖配置错误。

#### 2. 定时调度

可以根据实际需要, 设定任务的运行时间, 共有 5 种时间类型: 分钟、小时、日、周、月, 具体可精确到秒。比如日任务可选择每天的几点几分运行, 周任务可选择每周几的几点几分运行, 月任务也可选择每月第几天的几点几分运行。对于周任务和月任务, 通常选择定时调度的

方式。

### 3. 周期调度

可按照小时、日等时间周期运行任务，与定时调度的区别是无须指定具体的开始运行时间。比如离线数据处理的大多数日任务就是这种类型，任务根据依赖关系，按照调度树的顺序从上依次向下运行，每个周期的具体运行时间随着每天资源和上游依赖任务运行的总体情况会有所不同。

### 4. 手动运行

当生产环境需要做一些数据修复或其他一次性的临时数据操作时，可以选择手动运行的任务类型，在开发环境（IDE）中写好脚本后发布到生产环境，再通过手动触发运行。

### 5. 补数据

在完成数据开发的发布以后，有些表需要进行数据初始化，比如有些日增量表要补齐最近三年的历史数据，这时就需要用到补数据任务了。可以设定需要补的时间区间，并圈定需要运行的任务节点，从而生成一个补数据的工作流，同时还能选择并行的运行方式以节约时间。

### 6. 基线管理

基于充分利用计算资源，保证重点业务数据优先产出，合理安排各类优先级任务的运行，调度系统引入了按优先级分类管理的方法。优先级分类从 1~9，数字越大代表优先级越高，系统会先保障高优先级任务的运行资源。对于同一类优先级的任务，放到同一条基线中，这样可以实现按优先级不同进行分层的统一管理，并可对基线的运行时间进行预测估计，以监控是否能在规定的时间内完成。基线运行监控如表 4.1 所示。

表 4.1 基线运行监控

序号	基线名称	完成状态	基线时间 (预警及承诺)	最晚任务信息 及运行状态
1	基线 1	已完成 完成时间: 07-25 04:40 时间余量: 80 分钟 状态: 安全	预警: 07-25 06:00 承诺: 07-25 07:00	任务 ID1、任务名称 1, 执行状态: 完成, 责任人 1
2	基线 2	已完成 完成时间: a 时间余量: -b 状态: 破线	预警: c 承诺: d	任务 ID2、任务名称 2, 执行状态: 完成, 责任人 2
3	基线 3	未完成 预计完成时间: x 时间余量: -y (表示超出承诺时间 y 时长) 状态: 破线	预警: w 承诺: z	任务 ID3、任务名称 3, 执行状态: 运行中, 责任人 3
...	...	...	...	...

7. 监控报警

调度系统有一套完整的监控报警体系，包括针对出错的节点、运行超时未完成的节点，以及可能超时的基线等，设置电话、短信、邮件等不同的告警方式，实现了日常数据运维的自动化。具体产品介绍请参考“数据质量”章节。

阿里巴巴的大数据调度系统经过几年的迭代研发和实际运行，目前已经很好地支撑阿里系各个数据团队各类数据处理任务的日常运行和维护，每天在系统上运行的各类数据处理任务多达数十万个，日处理数据量达 PB 级，在性能、成本控制、稳定性等方面取得了良好的效果。

## 第5章

# 实时技术

在大数据系统中，离线批处理技术可以满足非常多的数据使用场景需求，但在 DT 时代，每天面对的信息是瞬息万变的，越来越多的应用场景对数据的时效性提出了更高的要求。数据价值是具有时效性的，在一条数据产生的时候，如果不能及时处理并在业务系统中使用，就不能让数据保持最高的“新鲜度”和价值最大化。

如图 5.1 所示是 2016 年“双 11”全球狂欢节当天，面向媒体开发的数据直播大屏在 24 点时定格的成交数据：1207 亿。在前台实时直播的数据，实际上是阿里实时计算系统在承载。直播大屏对数据有着非常高的精度要求，同时面临着高吞吐量、低延时、零差错、高稳定等多方面的挑战。在“双 11”的 24 小时中，支付峰值高达 12 万笔/秒，下单峰值达 17.5 万笔/秒，处理的总数据量高达百亿，并且所有数据是实时对外披露的，所以数据的实时计算不能出现任何差错。除此之外，所有的代码和计算逻辑都需要封存，并随时准备面对监管机构的问询和检查。

除面向媒体的数据大屏外，还有面向商家端的数据大屏、面向阿里巴巴内部业务运营的数据大屏。整个大屏直播功能需要实时处理的数据量非常庞大，每秒的总数据量更是高达亿级别，这就对实时计算架构提出了非常高的要求。面对如此庞大的数据量，阿里巴巴的实时处理是如

何做到高精度、高吞吐量、低延时、强保障的呢?



图 5.1 数据直播大屏

## 5.1 简介

相对于离线批处理技术,流式实时处理技术作为一个非常重要的技术补充,在阿里巴巴集团内被广泛使用。在大数据业界中,流计算技术的研究是近年来非常热门的课题。

业务诉求是希望能在第一时间拿到经过加工后的数据,以便实时监控当前业务状态并做出运营决策,引导业务往好的方向发展。比如网站上一个访问量很高的广告位,需要实时监控广告位的引流效果,如果转化率非常低的话,运营人员就需要及时更换为其他广告,以避免流量资源的浪费。在这个例子中,就需要实时统计广告位的曝光和点击等指标作为运营决策的参考。

按照数据的延迟情况,数据时效性一般分为三种(离线、准实时、实时):

- 离线:在今天( $T$ )处理 $N$ 天前( $T-N$ ,  $N \geq 1$ )的数据,延迟时



间粒度为天。

- 准实时：在当前小时 ( $H$ ) 处理  $N$  小时前 ( $H-N$ ,  $N>0$ , 如 0.5 小时、1 小时等) 的数据, 延迟时间粒度为小时。
- 实时：在当前时刻处理当前的数据, 延迟时间粒度为秒;

离线和准实时都可以在批处理系统中实现 (比如 Hadoop、MaxCompute、Spark 等系统), 只是调度周期不一样而已, 而实时数据则需要在流式处理系统中完成。简单来说, 流式数据处理技术是指业务系统每产生一条数据, 就会立刻被采集并实时发送到流式任务中进行处理, 不需要定时调度任务来处理数据。

整体来看, 流式数据处理一般具有以下特征。

### 1. 时效性高

数据实时采集、实时处理, 延时粒度在秒级甚至毫秒级, 业务方能够在第一时间拿到经过加工处理后的数据。

### 2. 常驻任务

区别于离线任务的周期调度, 流式任务属于常驻进程任务, 一旦启动后就会一直运行, 直到人为地终止, 因此计算成本会相对比较高。这一特点也预示着流式任务的数据源是无界的, 而离线任务的数据源是有界的。这也是实时处理和离线处理最主要的差别, 这个特性会导致实时任务在数据处理上有一定的局限性。

### 3. 性能要求高

实时计算对数据处理的性能要求非常严格, 如果处理吞吐量跟不上采集吞吐量, 计算出来的数据就失去了实时的特性。比如实时任务 1 分钟只能处理 30 秒采集的数据, 那么产出的数据的延时会越来越长, 不能代表当前时刻的业务状态, 有可能导致业务方做出错误的运营决策。在互联网行业中, 需要处理的数据是海量的, 如何在数据量快速膨胀的情况下也能保持高吞吐量和低延时, 是当前面临的重要挑战。因此, 实

时处理的性能优化占了任务开发的很大一部分工作。

#### 4. 应用局限性

实时数据处理不能替代离线处理，除了计算成本较大这个因素外，对于业务逻辑复杂的场景（比如双流关联或者需要数据回滚的情况），其局限性导致支持不足。另外，由于数据源是流式的，在数据具有上下文关系的情况下，数据到达时间的不确定性导致实时处理跟离线处理得出来的结果会有一定的差异。

## 5.2 流式技术架构

在流式计算技术中，需要各个子系统之间相互依赖形成一条数据处理链路，才能产出结果最终对外提供实时数据服务。在实际技术选型时，可选的开源技术方案非常多，但是各个方案的整体架构是类似的，只是各个子系统的实现原理不太一样。另外，流式技术架构中的系统跟离线处理是有交叉的，两套技术方案并不是完全独立的，并且在业界中有合并的趋势。

各个子系统按功能划分的话，主要分为以下几部分。

#### 1. 数据采集

数据的源头，一般来自于各个业务的日志服务器（例如网站的浏览行为日志、订单的修改日志等），这些数据被实时采集到数据中间件中，供下游实时订阅使用。

#### 2. 数据处理

数据被采集到中间件中后，需要下游实时订阅数据，并拉取到流式计算系统的任务中进行加工处理。这里需要提供流计算引擎以支持流式

任务的执行。

### 3. 数据存储

数据被实时加工处理（比如聚合、清洗等）后，会写到某个在线服务的存储系统中，供下游调用方使用。这里的写操作是增量操作，并且是源源不断的。

### 4. 数据服务

在存储系统上会架设一层统一的数据服务层（比如提供 HSF 接口、HTTP 服务等），用于获取实时计算结果。

整体技术架构如图 5.2 所示。

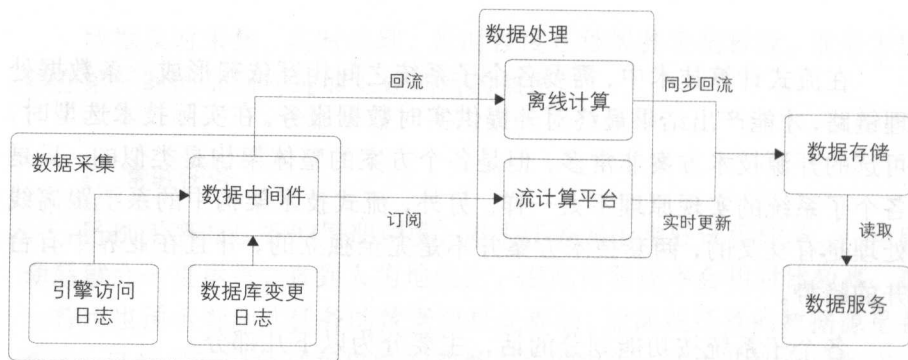


图 5.2 流式技术架构图

从图 5.2 可以看出，在数据采集和数据服务部分实时和离线是公用的，因为在这两层中都不需要关心数据的时效性。这样才能做到数据源的统一，避免流式处理和离线处理的不一致。

## 5.2.1 数据采集

数据采集是整个数据处理链路的源头，是所有数据处理链路的根节点，既然需要做到实时计算，那么自然就需要做到实时采集了。所采集

的数据都来自于业务服务器，从所采集的数据种类来看，主要可以划分为两种：

- 数据库变更日志，比如 MySQL 的 binlog 日志、HBase 的 hlog 日志、OceanBase 的变更日志、Oracle 的变更日志等。
- 引擎访问日志，比如用户访问网站产生的 Apache 引擎日志、搜索引擎的接口查询日志等。

不管是数据库变更日志还是引擎访问日志，都会在业务服务器上落地成文件，所以只要监控文件的内容发生变化，采集工具就可以把最新的数据采集下来。一般情况下，出于吞吐量以及系统压力上的考虑，并不是新增一条记录就采集一次，而是基于下面的原则，按批次对数据进行采集。

- 数据大小限制：当达到限制条件时，把目前采集到的新数据作为一批（例如 512KB 写一批）。
- 时间阈值限制：当时间达到一定条件时，也会把目前采集到的新数据作为一批，避免在数据量少的情况下一直不采集（例如 30 秒写一批）。

只要上面的其中一个条件达到了，就会被作为一批新数据采集到数据中间件中。这两个条件的参数需要根据业务的需求来设定，当批次采集频繁时，可以降低延时，但必然会导致吞吐量下降。

对于采集到的数据需要一个数据交换平台分发给下游，这个平台就是数据中间件。数据中间件系统有很多实现方式，比如开源的系统有 Kafka，而阿里巴巴集团内部用得比较多的是 TimeTunnel（原理和 Kafka 类似），还有 MetaQ、Notify 等消息系统。

从图 5.3 可以看出，消息系统是数据库变更节点的上游，所以它的延时比数据中间件低很多，但是其支持的吞吐量有限。因此，消息系统一般会用作业务数据库变更的消息中转，比如订单下单、支付等消息。对于其他较大的业务数据（每天几十 TB 的容量），一般会通过数据中间件系统来中转，虽然它的延时在秒级，但是其支持的吞吐量高。消息系统和数据中间件的性能对比如表 5.1 所示。

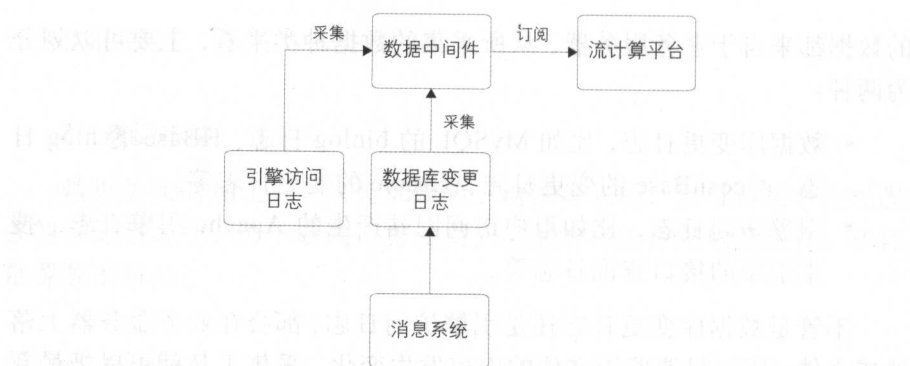


图 5.3 消息系统与数据中间件、数据库之间的关系

表 5.1 消息系统和数据中间件的性能对比

	时 效 性	吞 吐 量
消息系统	毫秒	低
数据中间件	秒	高

另外，在一些情况下，有些业务并没有通过消息系统来对数据库进行更新（比如有些子业务的订单数据是通过同步方式导入 MySQL 的）。也就是说，从消息系统中获取的数据并不是最全的，而通过数据库变更日志拿到的业务变更过程数据肯定是全的。因此，为了和离线数据源保持一致，一般都是通过数据中间件来采集数据库变更数据这种形式来获取实时数据的（这需要在数据处理层对业务主键进行 merge 处理，比如一笔订单可能会被变更多次，会有多条变更记录，这时就需要进行 merge 拿到最新的数据）。

时效性和吞吐量是数据处理中的两个矛盾体，很多时候需要从业务的角度来权衡使用什么样的系统来做数据中转。

## 5.2.2 数据处理

实时计算任务部署在流式计算系统上，通过数据中间件获取到实时源数据后进行实时加工处理。在各大互联网公司中，有各种开源的和非开源的流计算引擎系统在使用。在业界使用比较广泛的是 Twitter 开源

的 Storm 系统、雅虎开源的 S4 系统、Apache 的 Spark Streaming，以及最近几年兴起的 Flink。这些系统的整体架构大同小异，但是很多细节上的实现方式不太一样，适用于不同的应用场景。

在阿里巴巴集团内使用比较多的是阿里云提供的 StreamCompute 系统，作为业界首创的全链路流计算开发平台，涵盖了从数据采集到数据生产各个环节，力保流计算开发严谨、可靠。其提供的 SQL 语义的流式数据分析能力（StreamSQL），让流数据分析门槛不再存在。它在 Storm 的基础上包装了一层 SQL 语义，方便开发人员通过写 SQL 就可以实现实时计算，不需要关心其中的计算状态细节，大大提高了开发效率，降低了流计算的门槛。当然，它也支持传统模式的开发，就像 Hadoop 中的 Hive 和 MapReduce 的关系一样，根据不同的应用场景选择不同的方式。另外，StreamCompute 还提供了流计算开发平台，在这个平台上就可以完成应用的相关运维工作，不需要登录服务器操作，极大地提高了运维效率。

下面以 Storm 为例，简单讲一下流数据处理的原理。实时应用的整个拓扑结构是一个有向无环图（详情可参考 Apache Storm 的官网：<http://storm.apache.org/index.html>），如图 5.4 所示。

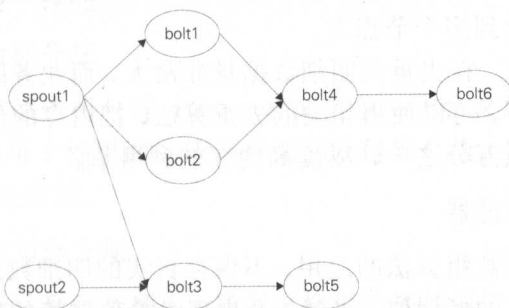


图 5.4 实时应用的整个拓扑结构图

- spout：拓扑的输入，从数据中间件中读取数据，并且根据自定义的分发规则发送给下游的 bolt，可以有多个输入源。
- bolt：业务处理单元，可以根据处理逻辑分为多个步骤，其相互之间的数据分发规则也是自定义的。

实时数据处理应用出于性能考虑，计算任务往往是多线程的。一般会根据业务主键进行分桶处理，并且大部分计算过程需要的数据都会放在内存中，这样会大大提高应用的吞吐量。当然，为了避免内存溢出，内存中过期的数据需要定时清理，可以按照 LRU（最近最少使用）算法或者业务时间集合归类清理（比如业务时间属于  $T-1$  的，会在今天凌晨进行清理）。

下面就实时任务遇到的几个典型问题进行讲解。

### 1. 去重指标

在 BI（商业智能）统计类实时任务中，对于资源的消耗有一类指标是非常高的，那就是去重指标。由于实时任务为了追求处理性能，计算逻辑一般都是在内存中完成的，中间结果数据也会缓存在内存中，这就带来了内存消耗过多的问题。在计算去重时，势必要把去重的明细数据保存下来，当去重的明细数据达到上亿甚至几十亿时，内存中放不下了，怎么办？这时需要分两种情况去看：

- 精确去重。在这种情况下，明细数据是必须要保存下来的，当遇到内存问题时，可以通过数据倾斜来进行处理，把一个节点的内存压力分到多个节点上。
- 模糊去重。在去重的明细数据量非常大，而业务的精度要求不高的情况下，可以使用相关的去重算法，把内存的使用量降到千分之一甚至万分之一，以提高内存的利用率。

#### (1) 布隆过滤器

该算法是位数组算法的应用，不保存真实的明细数据，只保存明细数据对应哈希值的标记位。当然，会出现哈希值碰撞的情况，但是误差率可以控制，计算出来的去重值比真实值小。采用这个算法存储 1 亿条数据只需要 100 多 MB 的空间。

适用场景：统计精度要求不高，统计维度值非常多的情况。比如统计全网各个商家的 UV 数据，结果记录数达到上千万条。因为在各个维度之间，布隆过滤器是可以共用的。



## (2) 基数估计

该算法也是利用哈希的原理,按照数据的分散程度来估算现有数集的边界,从而得出大概的去重值总和。这里估算的去重值可能比真实值大,也可能比真实值小。采用这个算法存储 1 亿条数据只需要几 KB 的内存。

适用场景:统计精度要求不高,统计维度非常粗的情况。比如整个大盘的 UV 数据,每天的结果只有一条记录。基数估计在各个维度值之间不能共用,比如统计全天小时的 UV 数据,就需要有 24 个基数估计对象,因此不适合细粒度统计的场景。

这两个算法可以在网上搜索到具体的实现细节,这里就不细讲了。

## 2. 数据倾斜

数据倾斜是 ETL 中经常遇到的问题,比如计算一天中全网访客数或者成交额时,最终的结果只有一个,通常应该是在一个节点上完成相关的计算任务。在数据量非常大的时候,单个节点的处理能力是有限的,必然会遇到性能瓶颈。这时就需要对数据进行分桶处理,分桶处理和离线处理的思路是一样的。

### (1) 去重指标分桶

通过对去重值进行分桶 Hash,相同的值一定会被放在同一个桶中去重,最后再把每个桶里面的值进行加和就得到总值,这里利用了每个桶的 CPU 和内存资源。

### (2) 非去重指标分桶

数据随机分发到每个桶中,最后再把每个桶的值汇总,主要利用的是各个桶的 CPU 能力。

## 3. 事务处理

由于实时计算是分布式处理的,系统的不稳定性必然会导致数据的处理有可能出现失败的情况。比如网络的抖动导致数据发送不成功、机



器重启导致数据丢失等。在这些情况下,怎么做到数据的精确处理呢?上面提到的几个流计算系统几乎都提供了数据自动 ACK、失败重发以及事务信息等机制。

- 超时时间:由于数据处理是按照批次来进行的,当一批数据处理超时时,会从拓扑的 spout 端重发数据。另外,批次处理的数据量不宜过大,应该增加一个限流的功能(限定一批数据的记录数或者容量等),避免数据处理超时。
- 事务信息:每批数据都会附带一个事务 ID 的信息,在重发的情况下,让开发者自己根据事务信息去判断数据第一次到达和重发时不同的处理逻辑。
- 备份机制:开发人员需要保证内存数据可以通过外部存储恢复,因此在计算中用到的中间结果数据需要备份到外部存储中。

上面的这些机制都是为了保证数据的幂等性。

### 5.2.3 数据存储

实时任务在运行过程中,会计算很多维度和指标,这些数据需要放在一个存储系统中作为恢复或者关联使用。其中会涉及三种类型的数据:

- 中间计算结果——在实时应用处理过程中,会有一些状态的保存(比如去重指标的明细数据),用于在发生故障时,使用数据库中的数据恢复内存现场。
- 最终结果数据——指的是通过 ETL 处理后的实时结果数据,这些数据是实时更新的,写的频率非常高,可以被下游直接使用。
- 维表数据——在离线计算系统中,通过同步工具导入到在线存储系统中,供实时任务来关联实时流数据。后面章节中会讲到维表的使用方式。

数据库分为很多种类型,比如关系型数据库、列式数据库、文档数据库等,那么在选择实时任务所使用的数据库时应该注意哪些特征呢?

前面提到实时任务是多线程处理的,这就意味着数据存储系统必须能够比较好地支持多并发读写,并且延时需要在毫秒级才能满足实时的

性能要求。在实践中，一般使用 HBase、Tair、MongoDB 等列式存储系统。由于这些系统在写数据时是先写内存再落磁盘，因此写延时在毫秒级；读请求也有缓存机制，重要的是多并发读时也可以达到毫秒级延时。

但是这些系统的缺点也是比较明显的，以 HBase 为例，一张表必须要有 rowkey，而 rowkey 是按照 ASCII 码来排序的，这就像关系型数据库的索引一样，rowkey 的规则限制了读取数据的方式。如果业务方需要使用另一种读取数据的方式，就必须重新输出 rowkey。从这个角度来看，HBase 没有关系型数据库方便。但是 HBase 的一张表能够存储几 TB 甚至几十 TB 的数据，而关系型数据库必须要分库分表才能实现这个量级的数据存储。因此，对于海量数据的实时计算，一般会采用非关系型数据库，以应对大量的多并发读写。

下面介绍在数据统计中表名设计和 rowkey 设计的一些实践经验。

## 1. 表名设计

设计规则：汇总层标识+数据域+主维度+时间维度

例如：dws\_trd\_slr\_dtr，表示汇总层交易数据，根据卖家（slr）主维度+0 点截至当日（dtr）进行统计汇总。

这样做的好处是，所有主维度相同的数据都放在一张物理表中，避免表数量过多，难以维护。另外，可以从表名上直观地看到存储的是什么数据内容，方便排查问题。

## 2. rowkey 设计

设计规则：MD5 + 主维度 + 维度标识 + 子维度 1 + 时间维度 + 子维度 2

例如：卖家 ID 的 MD5 前四位 + 卖家 ID + app + 一级类目 ID + ddd + 二级类目 ID。

以 MD5 的前四位作为 rowkey 的第一部分，可以把数据散列，让服务器整体负载是均衡的，避免热点问题。在上面的例子中，卖家 ID 属于主维度，在查数据时是必传的。每个统计维度都会生成一个维度标识，

以便在 rowkey 上做区分。

## 5.2.4 数据服务

实时数据落地到存储系统中后,使用方就可以通过统一的数据服务获取到实时数据。比如下一章将要讲到的 OneService,其好处是:

- 不需要直连数据库,数据源等信息在数据服务层维护,这样当存储系统迁移时,对下游是透明的。
- 调用方只需要使用服务层暴露的接口,不需要关心底层取数逻辑的实现。
- 屏蔽存储系统间的差异,统一的调用日志输出,便于分析和监控下游使用情况。

## 5.3 流式数据模型

数据模型设计是贯通数据处理过程的,流式数据处理也一样,需要对数据流建模分层。实时建模跟离线建模非常类似,数据模型整体上分为五层(ODS、DWD、DWS、ADS、DIM)。

由于实时计算的局限性,每一层中并没有像离线做得那么宽,维度和指标也没有那么多,特别是涉及回溯状态的指标,在实时数据模型中几乎没有。

整体来看,实时数据模型是离线数据模型的一个子集,在实时数据处理过程中,很多模型设计就是参考离线数据模型实现的。

下面从数据分层、多流关联、维表使用这三个方面来详细说明。

### 5.3.1 数据分层

在流式数据模型中,数据模型整体上分为五层。

## 1. ODS 层

跟离线系统的定义一样，ODS 层属于操作数据层，是直接从业务系统采集过来的最原始数据，包含了所有业务的变更过程，数据粒度也是最细的。在这一层，实时和离线在源头上是统一的，这样的好处是用同一份数据加工出来的指标，口径基本是统一的，可以更方便进行实时和离线间数据比对。例如：原始的订单变更记录数据、服务器引擎的访问日志。

## 2. DWD 层

DWD 层是在 ODS 层基础上，根据业务过程建模出来的实时事实明细层，对于访问日志这种数据（没有上下文关系，并且不需要等待过程的记录），会回流到离线系统供下游使用，最大程度地保证实时和离线数据在 ODS 层和 DWD 层是一致的。例如：订单的支付明细表、退款明细表、用户的访问日志明细表。

## 3. DWS 层

订阅明细层的数据后，会在实时任务中计算各个维度的汇总指标。如果维度是各个垂直业务线通用的，则会放在实时通用汇总层，作为通用的数据模型使用。比如电商网站的卖家粒度，只要涉及交易过程，就会跟这个维度相关，所以卖家维度是各个垂直业务的通用维度，其中的汇总指标也是各个业务线共用的。例如：电商数据的几大维度的汇总表（卖家、商品、买家）。

## 4. ADS 层

个性化维度汇总层，对于不是特别通用的统计维度数据会放在这一层中，这里计算只有自身业务才会关注的维度和指标，跟其他业务线一般没有交集，常用于一些垂直创新业务中。例如：手机淘宝下面的某个爱逛街、微淘等垂直业务。

## 5. DIM 层

实时维表层的数据基本上都是从离线维表层导出来的，抽取到在线系统中供实时应用调用。这一层对实时应用来说是静态的，所有的 ETL 处理工作会在离线系统中完成。维表在实时应用的使用中跟离线稍有区别，后面章节中会详细说明。例如：商品维表、卖家维表、买家维表、类目维表。

下面通过简单的例子来说明每一层存储的数据。

- ODS 层：订单粒度的变更过程，一笔订单有多条记录。
- DWD 层：订单粒度的支付记录，一笔订单只有一条记录。
- DWS 层：卖家的实时成交金额，一个卖家只有一条记录，并且指标在实时刷新。
- ADS 层：外卖地区的实时成交金额，只有外卖业务使用。
- DIM 层：订单商品类目和行业的对应关系维表。

整体的数据流向如图 5.5 所示。

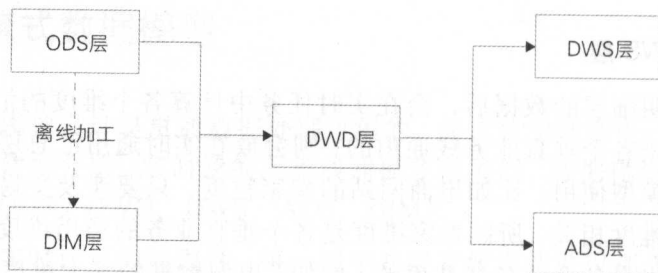


图 5.5 数据流向

其中，ODS 层到 DIM 层的 ETL 处理是在离线系统中进行的，处理完成后会同步到实时计算所使用的存储系统。ODS 层和 DWD 层会放在数据中间件中，供下游订阅使用。而 DWS 层和 ADS 层会落地到在线存储系统中，下游通过接口调用的形式使用。

在每一层中，按照重要性划分为 P0、P1、P2、P3 等级，P0 属于最高优先级保障。根据不同的优先级给实时任务分配不同的计算和存储资源，力求重要的任务可以得到最好的保障。

另外, 字段命名、表命名、指标命名是按照 OneData 规范来定义的, 以便更好地维护和管理。具体 OneData 的说明见后续章节。

### 5.3.2 多流关联

在流式计算中常常需要把两个实时流进行主键关联, 以得到对应的实时明细表。在离线系统中两个表关联是非常简单的, 因为离线计算在任务启动时已经可以获得两张表的全量数据, 只要根据关联键进行分桶关联就可以了。但流式计算不一样, 数据的到达是一个增量的过程, 并且数据到达的时间是不确定的和无序的, 因此在数据处理过程中会涉及中间状态的保存和恢复机制等细节问题。

比如 A 表和 B 表使用 ID 进行实时关联, 由于无法知道两个表的到达顺序, 因此在两个数据流的每条新数据到来时, 都需要到另外一张表中进行查找。如 A 表的某条数据到达, 到 B 表的全量数据中查找, 如果能查找到, 说明可以关联上, 拼接成一条记录直接输出到下游; 但是如果关联不上, 则需要放在内存或外部存储中等待, 直到 B 表的记录也到达。多流关联的一个关键点就是需要相互等待, 只有双方都到达了, 才能关联成功。

下面通过例子 (订单信息表和支付信息表关联) 来说明, 如图 5.6 所示。

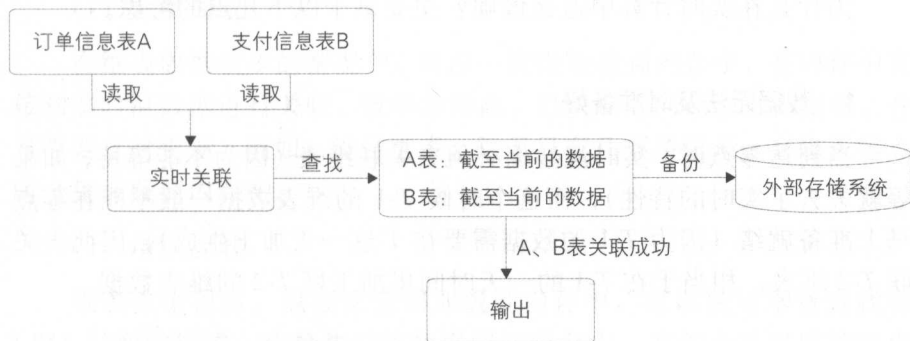


图 5.6 多流关联示例

在上面的例子中，实时采集两张表的数据，每到来一条新数据时都在内存中的对方表截至当前的全量数据中查找，如果能查找到，则说明关联成功，直接输出；如果没查找到，则把数据放在内存中的自己表数据集合中等待。另外，不管是否关联成功，内存中的数据都需要备份到外部存储系统中，在任务重启时，可以从外部存储系统中恢复内存数据，这样才能保证数据不丢失。因为在重启时，任务是续跑的，不会重新跑之前的数据。

另外，订单记录的变更有可能发生多次（比如订单的多个字段多次更新），在这种情况下，需要根据订单 ID 去重，避免 A 表和 B 表多次关联成功；否则输出到下游就会有多条记录，这样得到的数据是有重复的。

以上是整体的双流关联流程，在实际处理时，考虑到查找数据的性能，实时关联这个步骤一般会把数据按照关联主键进行分桶处理，并且在故障恢复时也根据分桶来进行，以降低查找数据量和提高吞吐量。

### 5.3.3 维表使用

在离线系统中，一般是根据业务分区来关联事实表和维表的，因为在关联之前维表的数据就已经就绪了。而在实时计算中，关联维表一般会使用当前的实时数据（ $T$ ）去关联  $T-2$  的维表数据，相当于在  $T$  的数据到达之前需要把维表数据准备好，并且一般是一份静态的数据。

为什么在实时计算中这么做呢？主要基于以下几点考虑。

#### 1. 数据无法及时准备好

当到达零点时，实时流数据必须去关联维表（因为不能等待，如果等就失去了实时的特性），而这个时候  $T-1$  的维表数据一般不能在零点马上准备就绪（因为  $T-1$  的数据需要在  $T$  这一天加工生成），因此去关联  $T-2$  维表，相当于在  $T-1$  的一天时间里加工好  $T-2$  的维表数据。

#### 2. 无法准确获取全量的最新数据

维表一般是全量的数据，如果需要实时获取到当天的最新维表数



据,则需要  $T-1$  的数据+当天变更才能获取到完整的维表数据。也就是说,维表也作为一个实时流输入,这就需要使用多流实时关联来实现。但是由于实时数据是无序的并且到达时间不确定,因此在维表关联上有歧义。

### 3. 数据的无序性

如果维表作为实时流输入的话,获取维表数据将存在困难。比如 10:00 点的业务数据成功关联维表,得到了相关的维表字段信息,这个时候是否就已经拿到最新的维表数据了呢?其实这只代表拿到截至 10:00 点的最新状态数据(实时应用永远也不知道什么时候才是最新状态,因为不知道维表后面是否会发生变更)。

因此在实时计算中维表关联一般都统一使用  $T-2$  的数据,这样对于业务来说,起码关联到的维表数据是确定的(虽然维表数据有一定的延时,但是许多业务的维表在两天之间变化是很少的)。

在有些业务场景下,可以关联  $T-1$  的数据,但  $T-1$  的数据是不全的。比如在  $T-1$  的晚上 22:00 点开始对维表进行加工处理,在零点到达之前,有两个小时可以把数据准备好,这样就可以在  $T$  的时候关联  $T-1$  的数据了,但是会缺失两个小时的维表变更过程。

另外,由于实时任务是常驻进程的,因此维表的使用分为两种形式。

#### (1) 全量加载

在维表数据较少的情况下,可以一次性加载到内存中,在内存中直接和实时流数据进行关联,效率非常高。但缺点是内存一直占用着,并且需要定时更新。例如:类目维表,每天只有几万条记录,在每天零点时全量加载到内存中。

#### (2) 增量加载

维表数据很多,没办法全部加载到内存中,可以使用增量查找和 LRU 过期的形式,让最热门的数据留在内存中。其优点是可以控制内存的使用量;缺点是需要查找外部存储系统,运行效率会降低。例如:会员维表,有上亿条记录,每次实时数据到达时,去外部数据库中查询,



并且把查询结果放在内存中，然后每隔一段时间清理一次最近最少使用的数据，以避免内存溢出。

在实际应用中，这两种形式根据维表数据量和实时性能要求综合考虑来选择使用。

## 5.4 大促挑战&保障

大促是电商行业的狂欢节，在这期间，各个业务系统面临的峰值都会达到最高点，每年大促的海量数据处理给实时计算的性能和保障提出了很大的挑战。

### 5.4.1 大促特征

大促和日常比较，在数据量以及要求上有非常大的区别，日常不怎么关注的点，在大促的时候会被放大，并且一天中的峰值特别明显，数据量是其他时间点的几倍甚至数十倍，这对系统的抗压能力要求非常高，不能因为洪流的到来而把系统压垮。

#### 1. 毫秒级延时

大促期间，业务方和用户都会对实时数据非常关注，特别是在跨过零点的时候，第一个实时数字的跳动对业务方来说意义重大，预示着大促狂欢节真正开始。其他的产品，例如全球媒体直播大屏，更是要求延时达到毫秒级。这种要求吞吐量和延时兼得的情况，必须要做一些有针对性的优化工作才能满足要求。

#### 2. 洪峰明显

大促就是全国乃至全世界的狂欢节，零点开售时的峰值陡峰是非常明显的，一般是日常峰值的几十倍，这对数据处理链路的每个系统都是

巨大的挑战。因此，在大促前会进行多次全链路压测和预案梳理，确保系统能够承载住洪峰的冲击。

### 3. 高保障性

由于关注的人非常多，只要出现数据延迟或者数据质量的问题，业务方的反弹就比较大，并且会第一时间感知到数据异常。因此，在大促期间一般都要求高保障性，一些特殊的情况甚至需要做到强保障。

对于强保障的数据，需要做多链路冗余（从采集、处理到数据服务整个数据链路都需要做物理隔离）（见图 5.7）。当任何一条链路出现问题时，都能够一键切换到备链路，并且需要对业务方透明，让下游感知不到有链路上的切换（由于各个链路计算的速度有一定的差异，会导致数据在切换时出现短时间下跌的情况，使用方需要做指标大小的判断，避免指标下跌对用户造成困扰）。

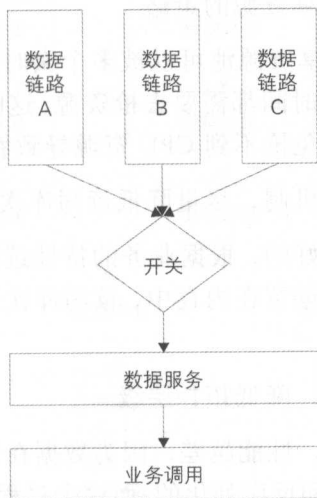


图 5.7 强保障数据多链路冗余示意图

### 4. 公关特性

大促期间，数据及时对公众披露是一项重要的工作，这时候要求实时计算的数据质量非常高。这里面涉及主键的过滤、去重的精准和口径

的统一等一系列问题,只有把每一个环节都做好才能保障和离线的数据一致。

大促是一场对数据计算的高吞吐量、低延时、高保障性、高准确性的挑战。

## 5.4.2 大促保障

### 1. 如何进行实时任务优化

大促前的优化工作在实时计算中显得尤为重要,如果吞吐量跟不上,也就失去了实时的特性。吞吐量不佳原因非常多,有些跟系统资源有关,有些跟实现方式有关,以下几点是实时任务优化中经常需要考虑的要素。

#### (1) 独占资源和共享资源的策略

在一台机器中,共享资源池可以被多个实时任务抢占,如果一个任务在运行时 80% 以上的时间都需要去抢资源,这时候就需要考虑给它分配更多的独占资源,避免抢不到 CPU 资源导致吞吐量急剧下降。

#### (2) 合理选择缓存机制,尽量降低读写库次数

内存读写性能是最好的,根据业务的特性选择不同的缓存机制,让最热和最可能使用的数据留在内存中,读写库次数降低后,吞吐量自然就上升了。

#### (3) 计算单元合并,降低拓扑层级

拓扑结构层级越深,性能越差,因为数据在每个节点间传输时,大部分是需要经过序列化和反序列化的,而这个过程非常消耗 CPU 和时间。

#### (4) 内存对象共享,避免字符拷贝

在海量数据处理中,大部分对象都是以字符串形式存在的,在不同线程间合理共享对象,可以大幅降低字符拷贝带来的性能消耗,不过要注意不合理使用带来的内存溢出问题。

### (5) 在高吞吐量和低延时间取平衡

高吞吐量和低延时这两个特性是一对矛盾体,当把多个读写库操作或者 ACK 操作合并成一个时,可以大幅降低因为网络请求带来的消耗,不过也会导致延时高一些,在业务上衡量进行取舍。

## 2. 如何进行数据链路保障

实时数据的处理链路非常长(数据同步→数据计算→数据存储→数据服务),每一个环节出现问题,都会导致实时数据停止更新。实时计算属于分布式计算的一种,而单个节点故障是常态的,这种情况在直播大屏中表现特别明显,因为数据不再更新,所有的用户都会发现数据出现了问题。因此,为了保障实时数据的可用性,需要对整条计算链路都进行多链路搭建,做到多机房容灾,甚至异地容灾(见图 5.8)。

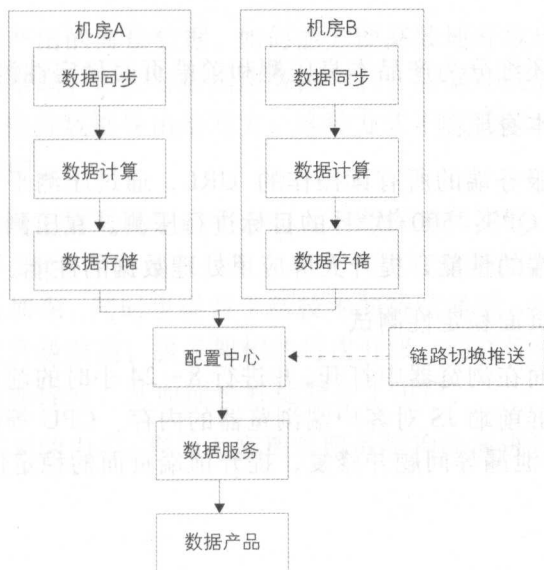


图 5.8 多机房容灾示意图

由于造成链路问题的情况比较多,并且一般不能在秒级定位到原因,因此会通过工具比对多条链路计算的结果数据,当某条链路出现问题时,它一定会比其他链路计算的值小,并且差异会越来越大。这时候

会一键切换到备链路，并且通过推送配置的形式让其秒级生效，所有的接口调用会立刻切换到备链路，对直播大屏完全透明，并且用户也感知不到故障的发生。

### 3. 如何进行压测

在大促备战中，会对实时链路进行多次压测，主要是模拟“双 11”的峰值情况，验证系统是否能够正常运行。压测都是在线上环境中进行的，分为数据压测和产品压测。

数据压测主要是蓄洪压测，就是把几个小时甚至几天的数据积累下来，并在某个时刻全部放开，模拟“双 11”洪峰流量的情况，这里面的数据是真实的。比如通过把实时作业的订阅数据点位调到几个小时或者几天前，这时候每一批读到的数据都是最多的，对实时计算的压力也最大。

产品压测还细分为产品本身压测和前端页面稳定性测试。

#### (1) 产品本身压测

收集大屏服务端的所有读操作的 URL，通过压测平台进行压测流量回放，按照 QPS：500 次/秒的目标进行压测。在压测过程中不断地迭代优化服务端的性能，提升大屏应用处理数据的性能。

#### (2) 前端页面稳定性测试

将大屏页面在浏览器中打开，并进行 8~24 小时的前端页面稳定性测试。监控大屏前端 JS 对客户端浏览器的内存、CPU 等的消耗，检测出前端 JS 内存泄漏等问题并修复，提升前端页面的稳定性。

## 第6章

# 数据服务

数据部门产出的海量数据，如何能方便高效地开放出去，是我们一直想要解决的难题。在没有数据服务的年代，数据开放的方式简单、粗暴，一般是直接将数据导出给对方。这种方式不仅低效，还带来了安全隐患等诸多问题。

为此，我们在数据服务这个方向上不断探索和实践。最早的数据服务雏形诞生于 2010 年，至今已有 7 个年头。在这期间，随着我们对业务的理解不断加深，同时也得益于新技术的持续涌现，对数据服务架构也进行了多次升级改造。服务架构的每次升级，均在性能、稳定性、扩展性等方面有所提升，从而能更好地服务于用户。

本章接下来的内容，将为大家揭示服务架构的演进过程以及详细的技术细节。

### 6.1 服务架构演进

阿里数据服务架构演进过程如图 6.1 所示。基于性能、扩展性和稳

定性等方面的要求，我们不断升级数据服务的架构，依次经历了内部代号为 DWSOA、OpenAPI、SmartDQ 和 OneService 的四个阶段，下面将详细介绍各个阶段的特点及问题。

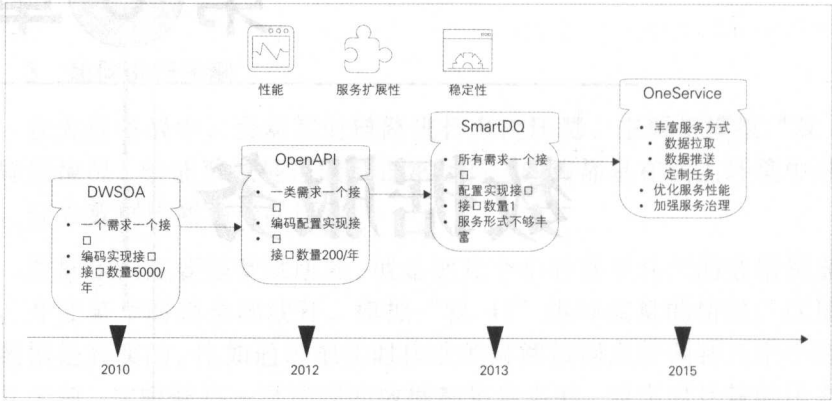


图 6.1 阿里数据服务架构演进过程

6.1.1 DWSOA

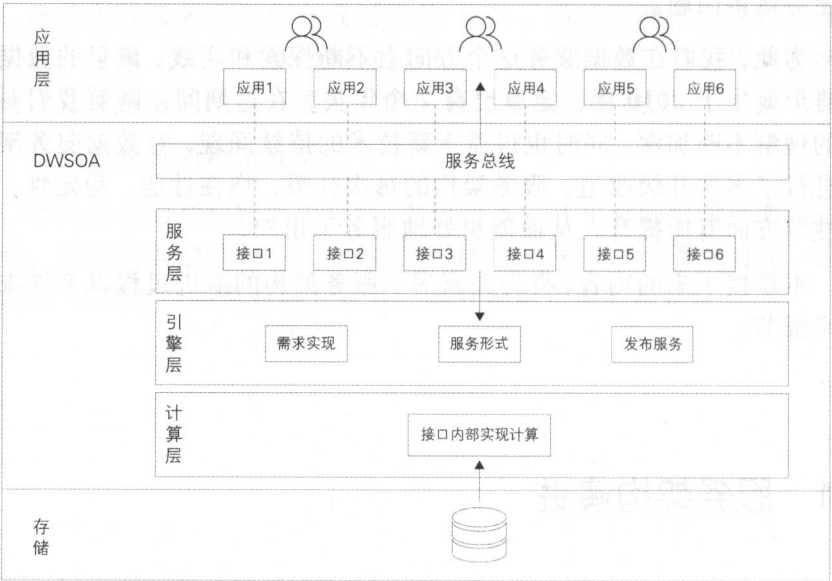


图 6.2 DWSOA 架构示意图

DWSOA 是数据服务的第一个阶段,也就是将业务方对数据的需求通过 SOA 服务的方式暴露出去。由需求驱动,一个需求开发一个或者几个接口,编写接口文档,开放给业务方调用。

这种架构实现起来比较简单,但是其缺陷也是特别明显的。一方面,接口粒度比较粗,灵活性不高,扩展性差,复用率低。随着业务方对数据服务的需求增加,接口的数量也会很快从一位数增加到两位数,从两位数增加到三位数,其维护成本可想而知。另一方面,开发效率不高,无法快速响应业务。一个接口从需求开发、测试到最终的上线,整个流程走完至少需要 1 天的时间,即使有时候仅仅是增加一、两个返回字段,也要走一整套流程,所以开发效率比较低,投入的人力成本较高。

### 6.1.2 OpenAPI

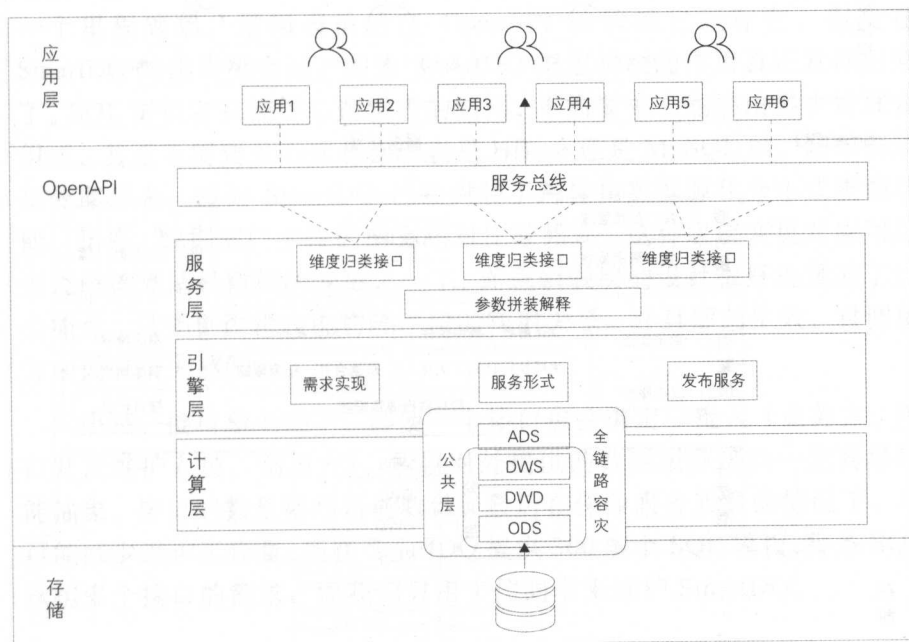


图 6.3 OpenAPI 架构示意图



DWSOA 阶段存在的明显问题，就是烟囱式开发，导致接口众多不好维护，因此需要想办法降低接口的数量。当时我们对这些需求做了调研分析，发现实现逻辑基本上就是从 DB 取数，然后封装结果暴露服务，并且很多接口其实是可以合并的。

OpenAPI 就是数据服务的第二个阶段。具体的做法就是将数据按照其统计粒度进行聚合，同样维度的数据，形成一张逻辑表，采用同样的接口描述。以会员维度为例：把所有以会员为中心的数据做成一张逻辑宽表，只要是查询会员粒度的数据，仅需要调用会员接口即可。通过一段时间的实施，结果表明这种方式有效地收敛了接口数量。

### 6.1.3 SmartDQ

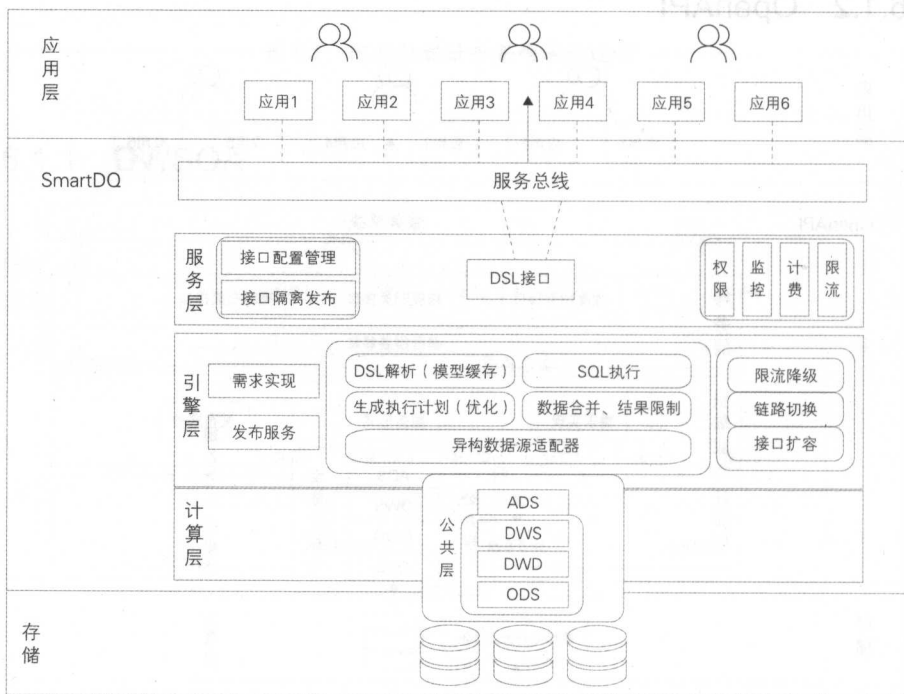


图 6.4 SmartDQ 架构示意图

然而，数据的维度并没有我们想象的那么可控，随着时间的推移，大家对数据的深度使用，分析数据的维度也越来越多，当时 OpenAPI 生产已有近 100 个接口；同时也带来大量对象关系映射的维护工作量。

于是，在 OpenAPI 的基础上，再抽象一层，用 DSL (Domain Specific Language, 领域专用语言) 来描述取数需求。新做一套 DSL 必然有一定的学习成本，因此采用标准的 SQL 语法，在此基础上做了一些限制和特殊增强，以降低学习成本。同时也封装了标准 DataSource，可以使用 ORM (Object Relation Mapping, 对象关系映射) 框架（目前比较主流的框架有 Hibernate、MyBatis 等）来解决对象关系映射问题。至此，所有的简单查询服务减少到只有一个接口，这大大降低了数据服务的维护成本。传统的方式查问题需要翻源码，确认逻辑；而 SmartDQ 只需要检查 SQL 的工作量，并且可以开放给业务方通过写 SQL 的方式对外提供服务，由服务提供者自己来维护 SQL，也算是服务走向 DevOps 的一个里程碑吧。逻辑表虽然在 OpenAPI 阶段就已经存在，但是在 SmartDQ 阶段讲更合适，因为 SmartDQ 把逻辑表的作用真正发挥出来了。SQL 提供者只需关心逻辑表的结构，不需要关心底层由多少物理表组成，甚至不需要关心这些物理表是 HBase 还是 MySQL 的，是单表还是分库分表，因为 SmartDQ 已经封装了跨异构数据源和分布式查询功能。此外，数据部门字段的变更相对比较频繁，这种底层变更对应用层来说应该算是最糟糕的变更之一了。而逻辑表层的设计很好地规避了这个痛点，只变更逻辑表中物理字段的映射关系，并且即刻生效，对调用方来说完全无感知。

**小结：**接口易上难下，即使一个接口也会绑定一批人（业务方、接口开发维护人员、调用方）。所以对外提供的数据服务接口一定要尽可能抽象，接口的数量要尽可能收敛，最后在保障服务质量的情况下，尽可能减少维护工作量。现在 SmartDQ 提供 300 多个 SQL 模板，每条 SQL 承担多个接口的需求，而我们只用 1 位同学来维护 SmartDQ。

## 6.1.4 统一的数据服务层

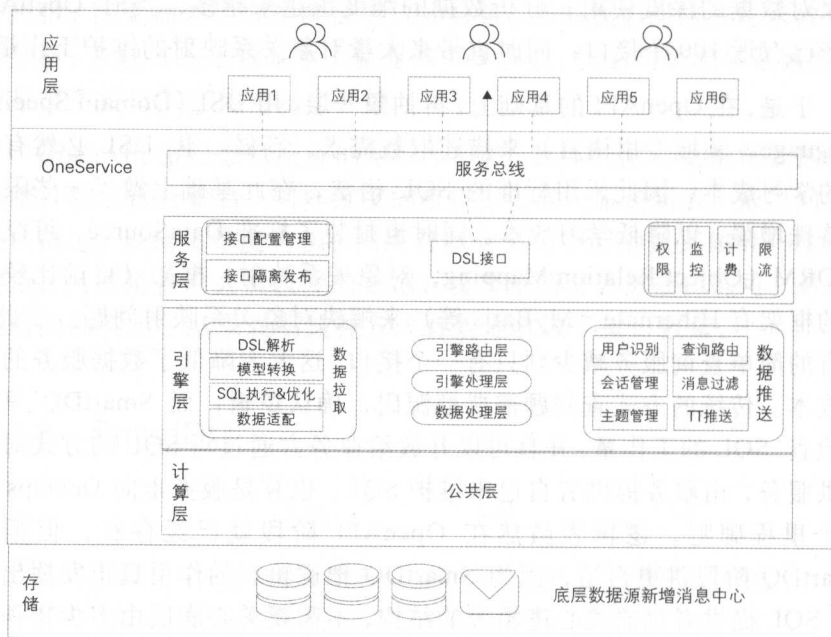


图 6.5 统一的数据服务层 (OneService) 架构示意图

第四个阶段是统一的数据服务层（即 OneService）。大家心里可能会有疑问：SQL 并不能解决复杂的业务逻辑啊。确实，SmartDQ 其实只满足了简单的查询服务需求。我们遇到的场景还有这么几类：个性化的垂直业务场景、实时数据推送服务、定时任务服务。所以 OneService 主要是提供多种服务类型来满足用户需求，分别是 OneService-SmartDQ、OneService-Lego、OneService-iPush、OneService-uTiming。

上面提到过，SmartDQ 不能满足个性化的取数业务场景，可以使用 Lego。Lego 采用插件化方式开发服务，一类需求开发一个插件，目前一共生产 5 个插件。为了避免插件之间相互影响，我们将插件做成微服务，使用 Docker 做隔离。

实时数据服务 iPush 主要提供 WebSocket 和 long polling 两种方式，其应用场景主要是商家端实时直播。在“双 11”当天，商家会迫不及

待地去刷新页面，在这种情况下 long polling 会给服务器带来成倍的压力。而 WebSocket 方式，可以在这种场景下，有效地缓解服务器的压力，给用户带来最实时的体验。

uTiming 主要提供即时任务和定时任务两种模式，其主要应用场景是满足用户运行大数据量任务的需求。

在 OneService 阶段，开始真正走向平台化。我们提供数据服务的核心引擎、开发配置平台以及门户网站。数据生产者将数据入库之后，服务提供者可以根据标准规范快速创建服务、发布服务、监控服务、下线服务，服务调用者可以在门户网站中快速检索服务，申请权限和调用服务。

## 6.2 技术架构

### 6.2.1 SmartDQ

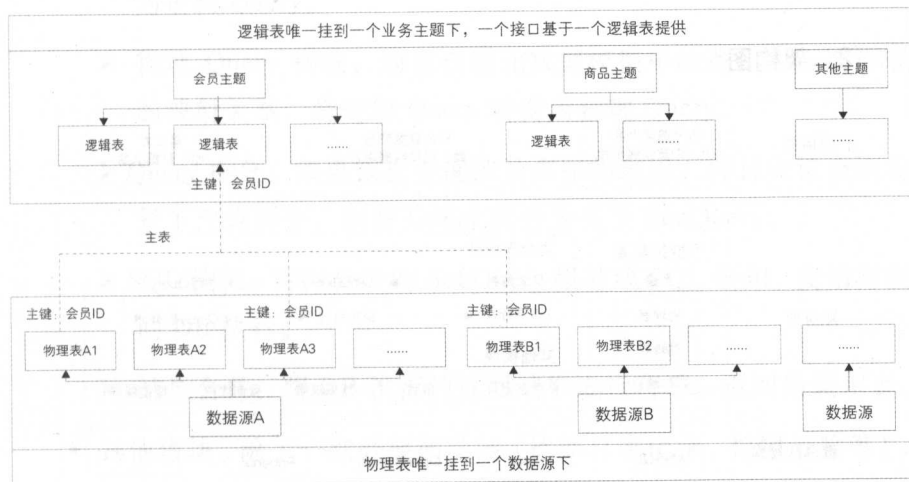


图 6.6 SmartDQ 的元数据模型架构示意图

#### 1. 元数据模型

SmartDQ 的元数据模型，简单来说，就是逻辑表到物理表的映射。

自底向上分别是：

### (1) 数据源

SmartDQ 支持跨数据源查询，底层支持接入多种数据源，比如 MySQL、HBase、OpenSearch 等。

### (2) 物理表

物理表是具体某个数据源中的一张表。每张物理表都需要指明主键由哪些列组成，主键确定后即可得知该表的统计粒度。

### (3) 逻辑表

逻辑表可以理解为数据库中的视图，是一张虚拟表，也可以看作是由若干主键相同的物理表构成的大宽表。SmartDQ 对用户展现的只是逻辑表，从而屏蔽了底层物理表的存储细节。

### (4) 主题

逻辑表一般会挂载在某个主题下，以便进行管理与查找。

## 2. 架构图

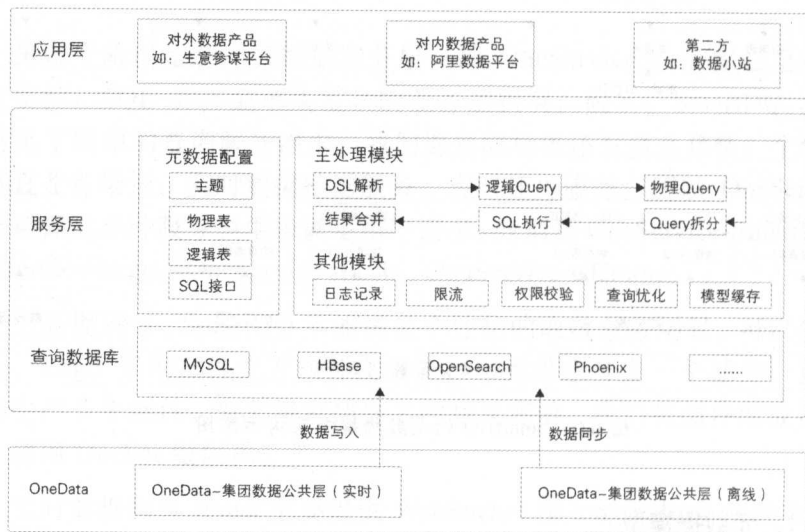


图 6.7 SmartDQ 架构示意图

### (1) 查询数据库

SmartDQ 底层支持多种数据源,数据的来源主要有两种:①实时公共层的计算作业直接将计算结果写入 HBase;②通过同步作业将公共层的离线数据同步到对应的查询库。

### (2) 服务层

- 元数据配置。数据发布者需要到元数据中心进行元数据配置,建立好物理表与逻辑表的映射关系,服务层会将元数据加载到本地缓存中,以便进行后续的模式解析。
- 主处理模块。一次查询从开始到结果返回,一般会经过如下几步。
  - DSL 解析:对用户的查询 DSL 进行语法解析,构建完整的查询树。
  - 逻辑 Query 构建:遍历查询树,通过查找元数据模型,转变为逻辑 Query。
  - 物理 Query 构建:通过查找元数据模型中的逻辑表与物理表的映射关系,将逻辑 Query 转变为物理 Query。
  - Query 拆分:如果该次查询涉及多张物理表,并且在该查询场景下允许拆分,则将 Query 拆分为多个 SubQuery。
  - SQL 执行:将拆分后的 SubQuery 组装成 SQL 语句,交给对应的 DB 执行。
  - 结果合并:将 DB 执行的返回结果进行合并,返回给调用者。
- 其他模块。除了一些必要的功能(比如日志记录、权限校验等),服务层中的一些模块是专门用于性能及稳定性方面的优化的,具体介绍请见 6.3 节的内容。

## 6.2.2 iPush

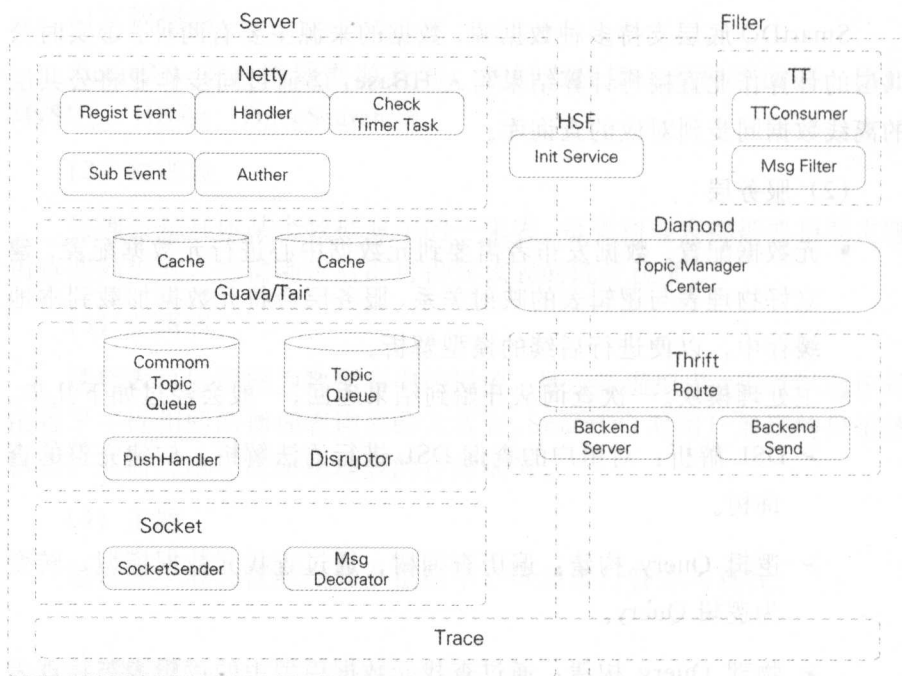


图 6.8 iPush 应用架构示意图

iPush 应用产品是一个面向 TT、MetaQ 等不同消息源，通过定制过滤规则，向 Web、无线等终端推送消息的中间件平台。iPush 核心服务器端基于高性能异步事件驱动模型的网络通信框架 Netty 4 实现，结合使用 Guava 缓存实现本地注册信息的存储，Filter 与 Server 之间的通信采用 Thrift 异步调用高效服务实现，消息基于 Disruptor 高性能的异步处理框架（可以认为是最快的消息框架）的消息队列，在服务器运行中 Zookeeper 实时监控服务器状态，以及通过 Diamond 作为统一的控制触发中心。



## 6.2.3 Lego

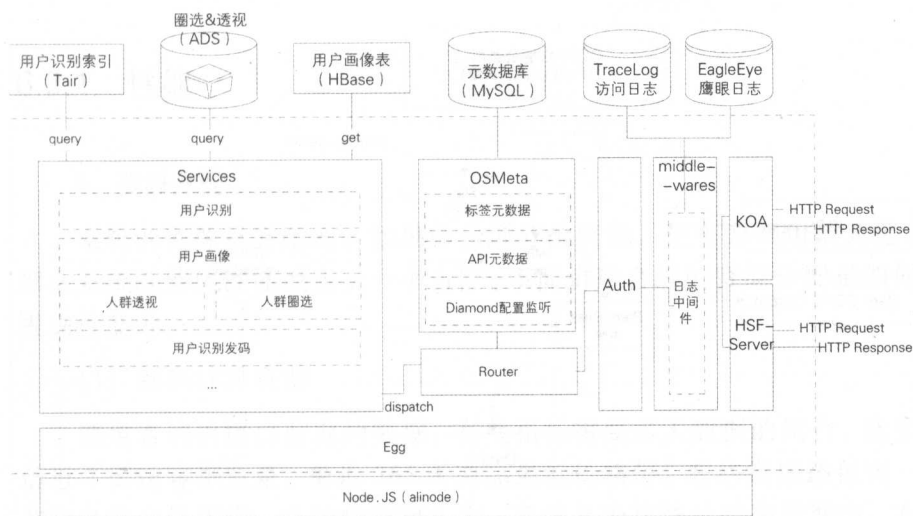


图 6.9 Lego 应用架构示意图

Lego 被设计成一个面向中度和高度定制化数据查询需求、支持插件机制的服务容器。它本身只提供日志、服务注册、Diamond 配置监听、鉴权、数据源管理等一系列基础设施，具体的数据服务则由服务插件提供。基于 Lego 的插件框架可以快速实现个性化需求并发布上线。

Lego 采用轻量级的 Node.JS 技术栈实现，适合处理高并发、低延迟的 IO 密集型场景，目前主要支撑用户识别发码、用户识别、用户画像、人群透视和人群圈选等在线服务。底层根据需求特点分别选用 Tair、HBase、ADS 存储数据。

## 6.2.4 uTiming

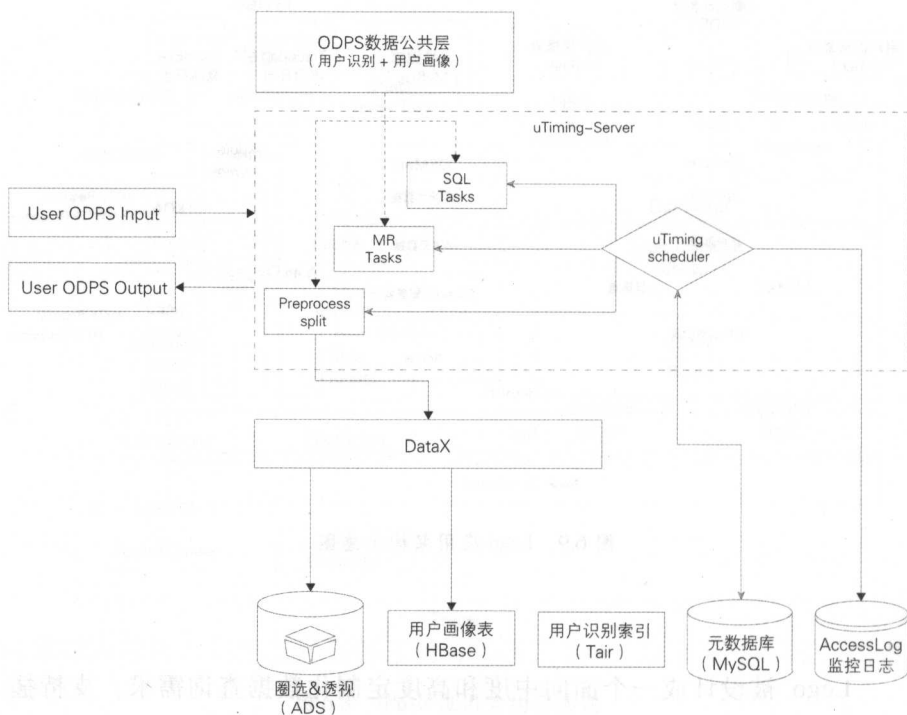


图 6.10 uTiming 应用架构示意图

uTiming 是基于在云端的任务调度应用，提供批量数据处理服务，支撑用户识别、用户画像、人群圈选三类服务的离线计算，以及用户识别、用户画像、人群透视的服务数据预处理、入库。

uTiming-scheduler 负责调度执行 SQL 或特定配置的离线任务，但并不直接对用户暴露任务调度接口。用户使用数据超市工具或 Lego API 建立任务。

## 6.3 最佳实践

### 6.3.1 性能

#### 1. 资源分配

系统的资源是有限的，如果能合理分配资源，使资源利用最大化，那么系统的整体性能就会上一个台阶。下面讲述合理的资源分配是如何提高性能的。

##### (1) 剥离计算资源

调用者调用接口获取的数据，有些指标需要多天数据的聚合，比如最近 7 天访客浏览量、最近 365 天商品最低价格等；有些指标还包含一些复杂的计算逻辑，比如成交回头率，其定义为在统计时间周期内，有两笔及以上成交父订单的买家数除以有成交父订单的买家数。

如此复杂的计算逻辑，如果放在每次调用接口时进行处理，其成本是非常高的。因此剥离复杂的计算统计逻辑，将其全部交由底层的数据公共层进行处理，只保留核心的业务处理逻辑。详细内容请参见第 9 章。

##### (2) 查询资源分配

查询接口分为两种：Get 接口，只返回一条数据；List 接口，会返回多条数据。一般来说，Get 查询基本都转换为 KV 查询，响应时间比较短，或者说查询代价比较小。而 List 查询的响应时间相对较长，且返回记录数比较多，这就增加了序列化以及网络传输的成本，查询代价肯定会更高一些。

假如将 Get、List 请求都放在同一个线程池中进行查询，那么查询效率会怎么样？想象一下如图 6.11 所示的场景，在高速公路上，行车道以及超车道全部都有大卡车在慢速行驶，后面的小轿车只能慢慢等待，并祈祷前方路段能少一些大卡车。这样整个路段的行车速度就降了下来，车流量也会下降许多。同理，虽然 Get 请求的真正查询耗时很短，但是会在队列等待上消耗大量的时间，这样整体的 QPS 会很不理想。

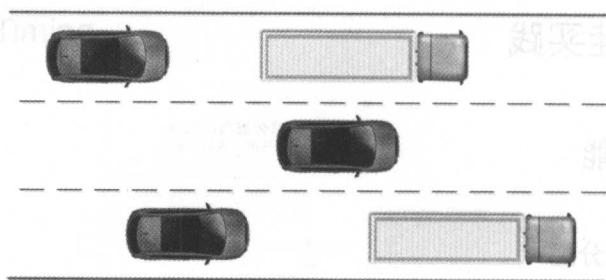


图 6.11 场景一

为此，我们设计了两个独立的线程池：Get 线程池和 List 线程池，分别处理 Get 请求和 List 请求，这样就不会因为某些 List 慢查询，而影响到 Get 快查询。系统的 QPS 比之前提升许多。回到上文的类比中，在高速公路上大卡车只行驶在最右车道上，小轿车行驶在其他车道上，这样整个路段也会畅通许多，如图 6.12 所示。

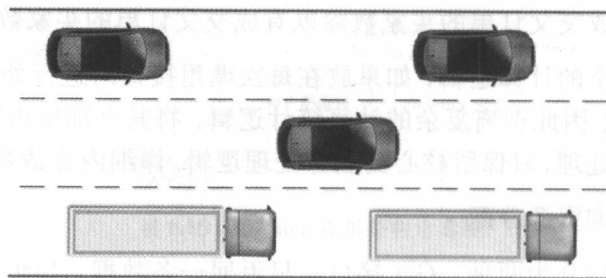


图 6.12 场景二

List 查询的响应时间相对较长，所以 List 线程池设置的最大运行任务数就稍微多一些。另外，由于超时的限制，List 线程池的等待队列不宜过长。具体的参数设置，可以根据压力测试的结果评估出来。后期，也可以根据线上调用日志的统计，比如 List 请求与 Get 请求的比例来进行优化调整。

### (3) 执行计划优化

① 查询拆分。举个例子，顾客去肯德基点餐，需要一个汉堡、一包薯条，再加一杯饮料。他可以先点个汉堡，拿到后再点包薯条，最后再

点杯饮料,是不是很浪费时间?为了节约时间,他可以叫上朋友来帮忙,每个人负责一样,同时去点餐。这样是快了很多,但是需要顾客付出额外的成本。那么现实中应该是怎么样的呢?顾客直接跟服务员说需要这些,服务员可以分工协作,最后统一放在餐盘中,告知顾客可以取餐了。

查询接口同样如此,接口暴露给调用者的指标都是逻辑字段,调用者不用关注这些逻辑字段对应的是哪张物理表的哪个物理字段。比如调用者调用了 A,B,C 三个指标,这些指标分别在三张物理表中,引擎层会将调用者的请求拆分成三个独立的查询,分别去三张物理表中查询,且这些查询是并发执行的。查询结束后,引擎层会将三个查询的结果汇总到一起返回给调用者,这样最大程度地降低了调用者的调用成本,并能保证查询性能(见图 6.13)。

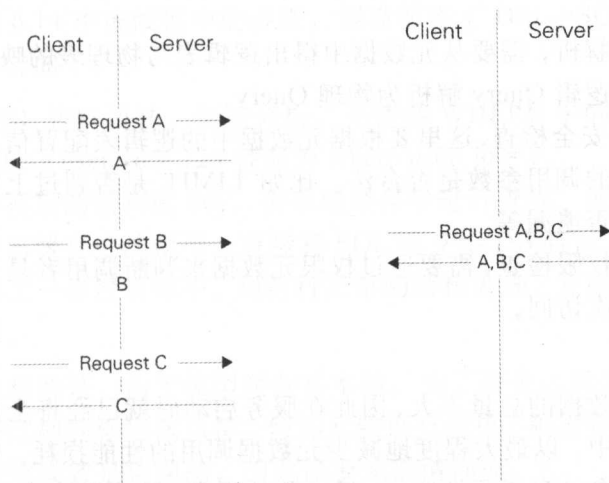


图 6.13 查询拆分

② 查询优化。上文提到 Get 请求与 List 请求分别有独立的线程池进行查询,但是一个请求具体是 Get 还是 List,则依赖调用者具体调用哪个方法。在很多情况下,调用者调用的方法不一定是合适的。比如,为了使代码更简洁,所有的调用全用 List 方法,这样就会造成很多本可以快速返回的查询,也在 List 线程池中进行排队。

查询优化,就是分析用户请求中的 SQL 语句,将符合条件的 List

查询转换为 Get 查询，从而提高性能。具体的步骤是：

- 解析 SQL 语句中的 WHERE 子句，提取出筛选字段以及筛选条件。
- 假如筛选字段中包含了该逻辑表的所有主键，且筛选条件都为 equal，则说明主键都已经确定为固定值，返回记录数肯定为 1 条。在这种场景中，List 查询就转换为 Get 查询。

## 2. 缓存优化

### (1) 元数据缓存

在接口查询的过程中，查询引擎需要频繁地调用元数据信息。举例来说：

- 查询解析，需要从元数据中得出逻辑表与物理表的映射关系，从而将逻辑 Query 解析为物理 Query。
- SQL 安全检查，这里要根据元数据中的逻辑表配置信息来检查调用者的调用参数是否合法。比如 LIMIT 是否超过上限、必传字段是否遗漏等。
- 字段权限检查，需要通过权限元数据来判断调用者是否有权限进行本次访问。
- .....

这些元数据的总量不大，因此在服务启动时就已经将全量数据加载到本地缓存中，以最大程度地减少元数据调用的性能损耗。后台对数据生产者的发布信息进行监听，一旦有新的发布，就重新加载一次元数据。不过，这时候的加载与初始化时不同，是一次增量更新，只会加载刚刚修改的元数据。

### (2) 模型缓存

接口查询的输入其实是 DSL，而最终提交给 DB 执行的是物理 SQL。在从 DSL 到物理 SQL 的转换过程中，经过了多步解析处理，如图 6.14 所示是 SmartDQ 架构图（见图 6.7）的一部分，展示了主处理模块的处理步骤。

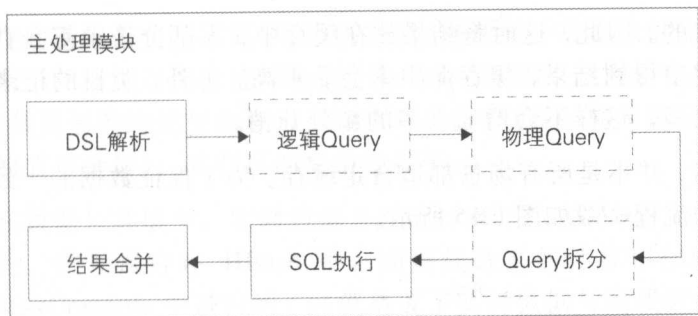


图 6.14 主处理模块处理步骤

模型缓存，就是将解析后的模型（包括逻辑模型、物理模型）缓存在本地。下次再遇到相似的 SQL 时，直接从缓存中得到解析结果，直接省略了图 6.14 中虚线框中的步骤，因而节省了 DSL->SQL 的解析时间。具体做法如下：

- 对 DSL 进行语法、词法分析，并替换 WHERE 中的常量。比如将 `where user_id = 123` 替换为 `where user_id = ?`。
- 以替换后的语句做 key，去本地缓存中进行查找。如果命中，则提取出缓存中的模型，直接将 SQL 提交给 DB 查询。
- 如果上一步没有命中，则进行正常的解析处理，并缓存解析后的结果。

需要注意的是，由于模型缓存在本地，为了避免占用太多的内存，需要定期将过期的模型淘汰掉。假如元数据有变更，则缓存中的模型有可能已经失效或者是错误的，因此需要全部清理掉。

### (3) 结果缓存

在某些场景下，会对查询结果进行缓存，以提高查询性能。例如：

- 某些查询可能比较复杂，直接查询 DB 响应时间较长。这时可以将结果进行缓存，下次执行相同的查询时，即可直接从缓存中获取结果，省去了 DB 查询这一步耗时操作。
- 还有一种场景，比如获取某个卖家所属类目的统计指标，一个类目下可能会有十几万个卖家，这些卖家请求的结果肯定是完全一



致的。因此，这时将结果放在缓存中，大部分请求都会直接从缓存中得到结果，缓存命中率会非常高。另外，类目的记录数不会太多，这样不会增加太多的额外开销。

当然，并不是所有场景都适合走缓存。为了保证数据的一致性，使用缓存的流程一般如图 6.15 所示。

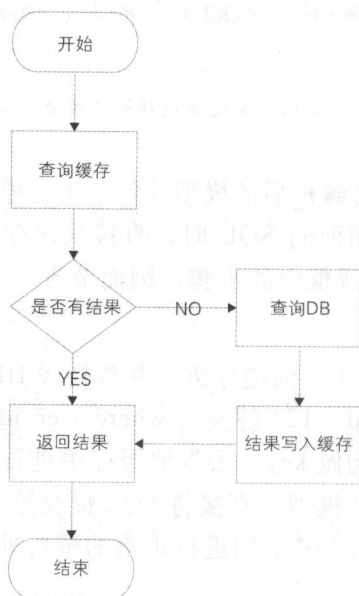


图 6.15 使用缓存流程图

假设有这样的场景：获取某个卖家对应的指标。由于每个卖家只能请求自己的指标，因此就会导致绝大部分请求都需要从 DB 查询，再写入缓存中。这样不仅使得单次请求的成本会提升，而且缓存的记录数会非常大，利用率也非常低。所以，这种场景其实是不太适合走缓存的，直接走 DB 查询是比较合适的。

### 3. 查询能力

#### (1) 合并查询

数据产品的有些场景，虽然表面上看只是展现几个数字而已，但是

后台的处理逻辑其实并不简单。举例来说,展现某一日卖家的支付金额,有个日期选择框可以任意选择日期。日期为今天时,展现的是实时数据(从零点截至当前的成交金额);日期为昨天时,展现的就是离线数据(最近1天的成交金额)。其背后的复杂性在于:

- 在数据公共层中,实时数据是在流计算平台 Galaxy 上进行计算的,结果保存在 HBase 中;而离线数据的计算和存储都是在 MaxCompute 中进行的。这就造成了实时数据与离线数据存储在两个数据源中,调用者的查询方式完全不同。
- 离线数据的产出时间,取决于上游任务的执行时间,以及当前平台的资源情况。所以其产出时间是无法估算的,有可能 3:00 产出,也有可能延迟到 6:00。在昨天的离线任务产出之前,其前台展现的数字只能来源于实时数据。
- 出于对性能和成本的考虑,实时作业做了一些折中,去重时,视情况可能使用一些不精准的去重算法,这就导致实时数据的计算结果与离线数据存在一些差异。

综上所述,离线数据最准确,需要优先使用离线数据。如果离线数据还未产出,则改用实时数据。所以在简单的数字背后,需要使用者清晰地了解上述三点。

为了降低这种场景的复杂性,我们设计了一种新的语法——REPLACE,如图 6.16 所示。

```
SELECT /*+replace_key(stat_date,user_id)*/
stat_date,user_id,pay_ord_amt_id_001 as amt
FROM do_tao_mbr_d
WHERE stat_date = ? and user_id = ?
REPLACE
SELECT
ddd as stat_date,seller as user_id,pay_ord_amt_dtr_001 as amt
FROM do_tao_other_1007
WHERE ddd = ? and seller = ? and app = 92495
```

图 6.16 REPLACE 语法

REPLACE 的效果就是用上边 SQL 的结果,根据 replace\_key 去替换下边 SQL 的结果。比如上述 SQL,上边的查询是取离线数据,下边

的查询是取实时数据，那么结果就是优先取离线数据，如果没有再去取实时数据。

调用者使用这样的语法，就可以实现离线数据替换实时数据的功能，不再需要考虑离线数据未产出等问题。

## (2) 推送服务

有些数据产品需要展现实时指标，为了追求数据的实时性，都是轮询请求最新数据。轮询的间隔时间设置很重要，如果设置间隔时间较长，用户体验会不太好；如果设置很短，对服务器的请求压力会非常大，从而影响整体性能。另外，这种轮询请求的方式，其实很大部分时间是在浪费资源，因为有可能后台的数据根本没有更新，而前端却一直在请求。那能不能换种方式呢？监听数据提供者，新数据产生时能够及时知道，并且告知用户，为此“推送”应运而生。推送服务很好地解决了数据更新的实时性问题，同时也减少了对服务器的请求压力。其主要从网络、内存、资源等方面做了如下设计：

- 对消息生产者进行监听。比如监听消息源 TT，一天的消息量可能有几百亿，但实际在线用户关心的可能就几亿甚至更少，所以并不是所有的消息都需要关心，做好消息过滤是非常必要的。
- 过滤后的消息量也是可观的，推送服务无法满足高效的响应需求，这就需要考虑将符合条件的消息放置在临时队列中，但对于有锁的队列，存在竞争则意味着性能或多或少会有些下降，所以采用无锁的队列 `Disruptor` 来存放消息是最佳的选择。在采用 `Disruptor` 的情况下，推送应用也考虑到可以对重要的消息配置单独的队列单线程运转，以提高性能。
- 消息的推送必须基于 `Socket` 来实现，`Netty` 在性能表现上比较优秀，采用基于高性能异步事件的网络通信框架 `Netty` 是我们的最终选择。不同事件采用不同的监听处理，职责分明也是提高性能的基础。
- 推送应用是典型的 IO 密集型系统，在采用多线程解决性能问题的同时，也带来了上下文切换的损耗。在注册消息向 `Filter` 广播时，采用协程方式可以大大减少上下文切换，为性能的提高做出

相应的贡献。

- 从业务角度出发,主题也会存在重要级别或者优先级,适当地控制线程数以及流量,为某些重要的业务消息节约服务器资源也是备选方案。
- 缓存的利用在推送应用中多处体现。例如对注册的在线用户信息做本地缓存,可以极大地提高读性能。
- 对突发事件的推送也有针对性地做了很多工作。比如过滤服务器异常重启时,在线用户信息需要重新向该过滤服务器投递,但每条用户信息才几百字节,如果逐条投递,则会造成高流量带宽的浪费,所以批量投递甚至打包投递会大大降低网络开销。

## 6.3.2 稳定性

### 1. 发布系统

上文中提到,服务启动时会将元数据全量加载到本地缓存中。数据生产者会对元数据做一些修改,并发布到线上。那么,如何保证用户的变更是安全的,不会导致线上故障呢?如何保证不同用户发布的变更不会相互影响呢?下面将会阐述发布系统在稳定性保障方面的作用。

#### (1) 元数据隔离

一般的应用都会有三个环境:日常环境、预发环境和线上环境。日常环境用于线下开发测试。预发环境隔离了外部用户的访问,用于在正式发布前校验即将上线的代码。为了保障系统的稳定性,根据应用环境设计了三套元数据:日常元数据、预发元数据和线上元数据,如图 6.17 所示。

由图 6.17 可知,三套元数据分别对应着三个应用环境,每个环境的应用只会访问对应的元数据。从用户修改元数据到最终正式上线,经过如下几步:

- 用户在元数据管理平台上进行操作,修改元数据。此时,DB 中的预发元数据发生了变更,但是还没有加载到本地缓存中。

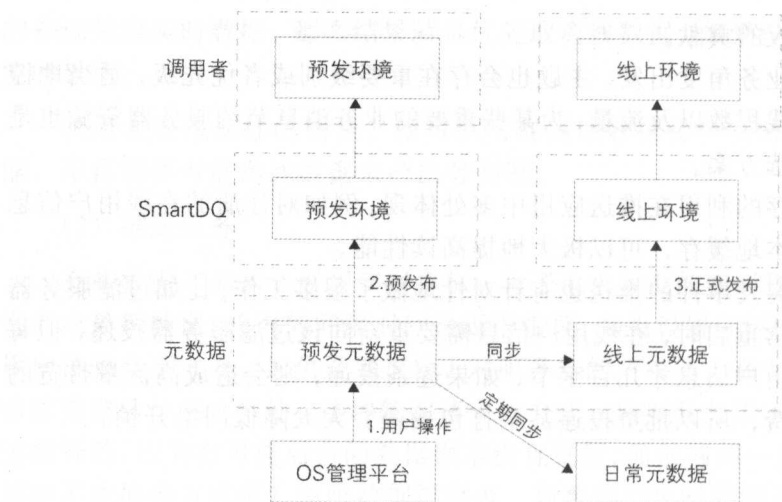


图 6.17 不同应用环境下的三套元数据

- 用户点击“预发布”，此时预发元数据就会被加载到引擎的本地缓存中，在预发环境中就可以看到用户的最新修改了。此时，可以验证用户的修改是否会影响线上已有的功能。
- 如果验证通过，则表明此次用户的修改是安全的。用户点击“正式发布”，预发元数据会将变更同步到线上元数据，并加载到引擎的本地缓存中。此时，在线上环境中也可以看到用户的变更。至此，发布的整个流程就结束了。

此外，会有一个定时任务，定期将预发元数据同步到日常环境。

通过元数据的隔离，使得用户的变更可以在预发环境中进行充分的验证，验证通过后再发布到线上环境中，避免了因用户误操作而导致线上故障，保障了系统的稳定性。

## (2) 隔离发布

发布系统还需要考虑到一点，就是隔离发布，即不同用户的发布不会相互影响。要实现这一点，需要做到：

- 资源划分。为了做到隔离发布，首先需要确定隔离的最小单元。由于调用者的查询请求最终都会转换成对某张逻辑表的查询，因

此我们决定将隔离的粒度控制在逻辑表层面上。

- 资源独占。当用户开始修改的时候，系统会锁定其正在修改的逻辑表及其下挂的物理表等资源，禁止其他用户修改。当用户正式发布变更后，就会释放锁定的资源，这时其他用户才可以对相关元数据进行修改。
- 增量更新。用户每次只会修改某张逻辑表的对应元数据，因此发布时引擎是不需要重新加载全量元数据的，只需要加载所发布的逻辑表元数据即可。同理，预发元数据与线上元数据之间的数据同步，也仅仅需要同步用户修改的部分。

## 2. 隔离

隔离的一个作用是将系统划分为若干个独立模块，当某个模块出现问题时，整体功能仍然能保证可用。隔离的另一个作用是可以对系统资源进行有效的管理，从而提高系统的可用性。

### (1) 机房隔离

将服务器部署在两个机房中，每个机房独立部署一个集群，且机器数量尽量保持均衡，以实现双机房容灾。当一个机房发生故障时，另一个机房中的应用仍然可以对外服务。同时，需要保障内部调用优先，服务调用同机房优先，最大程度地减少双机房部署带来的网络开销。

### (2) 分组隔离

不同调用者的优先级不尽相同，且查询场景也存在一定的差异。所以，可以根据某些条件将调用者进行分层，然后将服务端的机器划分为若干个分组，每个分组都有明确的服务对象和保障等级。即使某个分组出现性能较差的查询，或者有突发大流量涌入，也不会影响其他分组的正常使用。另外，可以动态地调整分组规则，以重新分配每个分组的机器数量，在总体机器数量不变的情况下，实现资源的最大化利用。

## 3. 安全限制

对调用者的调用做了诸多安全限制，以防止查询消耗大量的资源，

或者返回太多的记录。主要体现在以下几点：

- 最大返回记录数。数据库的查询强制带上 LIMIT 限制，具体的数值以用户配置为准。
- 必传字段。每张逻辑表都会配置主键，并标识哪些字段是调用者必须传入的。这样最终的 SQL 肯定会带上这些字段的限制条件，防止对表做全表扫描。
- 超时时间。设置合适的超时时间，以使得超时的查询能及时终止并释放资源，保障系统不会被偶发的超时拖垮。

#### 4. 监控

##### (1) 调用日志采集

如果要对调用做监控，首先要保证调用日志的完整性。对于每次调用都进行了采集，采集的信息包括：

- 基础信息，包括调用时间、接口名、方法名、返回记录数等。
- 调用者信息，包括调用者应用名、来源 IP 地址等。
- 调用信息，包括调用指标、查询筛选条件等。
- 性能指标，包括响应时间、是否走缓存等。
- 错误信息，包括出错原因、错误类型、数据源、错误堆栈等。

##### (2) 调用监控

有了调用日志，就可以监控系统的健康状况，及时发现问题。监控可以从以下几个方面展开：

- 性能趋势。总体的 QPS 趋势图、RT 趋势图、响应时长区间分布。分组性能统计、单机 QPS 统计，以对当前系统容量做评估。
- 零调用统计。找出最近  $N$  天无调用的表，进行下线处理，节约成本。
- 慢 SQL 查找。找出响应时间较长的 SQL，及时进行优化。
- 错误排查。当系统的调用错误数突增时，能从错误日志中及时发现出错原因、出错的数据源等。



## 5. 限流、降级

系统的总体容量，主要是根据平日的性能监控，以及定期的全链路压测评估得出，但是难免会遇到突发流量涌入的情况。此时，系统需要有合适的方式来应对突增流量，以免系统被压垮。

### (1) 限流

限流有很多种方法，我们采用的是应用内的 QPS 保护。针对调用者以及数据源等关键角色做了 QPS 阈值控制。也就是说，如果某个调用者的调用量突增，或者对某个数据源的查询流量突增，超过了预设的 QPS 阈值，则后续的请求立即失败返回，不再继续处理。通过快速失败，将超出系统处理能力的流量直接过滤掉，保障了系统的可用性。

### (2) 降级

查询引擎底层是支持多种数据源接入的，但是接入的数据源越多，系统就越复杂，出问题的概率也就越大。假设某个数据源突然出现问题，或者某个数据源中的某张表访问超时，那么该如何处理才能保障整体的可用性呢？

理想的做法肯定是将这些数据源、表全部隔离成独立的模块，单个模块的故障不会引起整体不可用。但是，实际中隔离带来的成本也是比较大的，且有可能造成资源的浪费。假如没有隔离措施，所有数据源共享资源，这时候就需要通过降级将故障影响降到最低。

降级主要有两种做法：

- 通过限流措施，将 QPS 置为 0，则对应的所有访问全部立即失败，防止了故障的扩散。
- 通过修改元数据，将存在问题的资源置为失效状态，则重新加载元数据后，对应的访问就全部失败了，不会再消耗系统资源。



## 第7章

# 数据挖掘

### 7.1 数据挖掘概述

进入 DT 时代，阿里巴巴作为全球最大的零售电子商务平台，正推动着大规模数据采集、计算、挖掘和产品化服务的生态圈构建。2016 年财报显示，阿里巴巴集团平台成交额突破 3 万亿元，年度活跃买家达到 4.23 亿，已成为全球最大的移动经济实体。与 Google、Facebook、Amazon 等世界上其他先进的互联网公司一样，高速增长的业务必然催生大数据挖掘应用的蓬勃发展。当我们从业务系统中能够轻松采集到海量数据时，往往会发现里面的有效数据信息却越来越稀疏，有效数据和无效数据的增长率是不成比例的。因此，如何从海量数据中挖掘出有效信息形成真正的生产力，是所有大数据公司需要面对的共同课题。

数据挖掘技术与数据仓储及计算技术的发展是相辅相成的，没有数据基础设施的发展与分布式并行计算技术，就不会有深度学习，更不会见证 AlphaGo 的神奇。同样在阿里巴巴集团，得益于阿里云 MaxCompute 云计算平台的发展，海量、高速、多变化、多终端的结构与非结构化数

据得以存储并高效地计算。近年来,阿里巴巴数据挖掘应用也呈现出井喷式的增长态势。面向海量会员和商品的全局画像、基于自然人的全域 ID-Mapping、广告精准投放平台、千人千面的个性化搜索与推荐技术、非人流量与恶意设备的识别、商业竞争情报的自动化挖掘系统……这些或传统或新兴的大数据挖掘应用已深入阿里巴巴业务的各个环节,“无数据不智能,无智能不商业”,大数据与 AI/机器学习融合后的新商业革命已然到来。

基于大数据的企业级数据挖掘需要包含两个要素:①面向机器学习算法的并行计算框架与算法平台;②面向企业级数据挖掘的算法资产管理体系。基于此,在接下来的章节中,我们会首先介绍阿里巴巴算法平台的发展历史,以及当下正在使用的算法计算框架和平台。然后阐述阿里巴巴的算法资产管理体系,领略企业级数据挖掘服务的标准、规范及资产管理艺术。最后将展示阿里巴巴当下最热门的一些数据挖掘应用案例,如消费者画像和互联网反作弊等,希望通过这些典型的应用案例,能让你更深层次地感知大数据、机器学习、人工智能融合后所产生的巨大能量及其在商业中发挥的巨大价值。

## 7.2 数据挖掘算法平台

2012 年以前,由于数据的规模还不是特别庞大,大部分挖掘应用所需处理的样本量在百万以内,而处理的特征一般也少于 100 维,那时业界有许多成熟的商业挖掘软件如 SAS、SPSS、Clementine 等,这些单机版运行的软件已经能满足绝大部分挖掘应用的需求,因此早几年阿里巴巴尚没有研发面向海量数据的高性能并行计算的算法平台。然而,随着数据量爆炸式的增长,以及分布式计算 Hadoop、Spark、Storm 等技术的引入,阿里巴巴的商业挖掘应用也步入大数据时代,和早几年不同,如今的挖掘算法需要面对的训练数据量动辄上亿,特征维度动辄百万,因此,近两、三年来,阿里巴巴集团也在大力发展自己的机器学习算法平台,并已取得阶段性成果。如今,阿里巴巴已建成一套稳定、高

效的算法平台，该平台架构于阿里云 MaxCompute、GPU 等计算集群之上，汇集了阿里巴巴集团大量优质的分布式算法，包括数据处理、特征工程、机器学习算法、文本算法等，可高效地完成海量、亿级维度数据的复杂计算。除此之外，平台还提供了一套极易操作的可视化编辑页面，大大降低了数据挖掘的门槛，提高了建模效率。未来还将面向外部客户开放，配合阿里云的其他基础数据设施，为外部企业提供数据挖掘应用的基础能力。

下面简单介绍阿里巴巴算法平台的框架和原理。

支持海量样本的高维特征训练是算法平台的必备要素，因此计算框架的选择非常重要。近几年来，业界主流的并行计算框架主要有 MapReduce、MPI、Spark 等。在阿里巴巴集团内部，基于 MapReduce 与 Hive 的计算已经能解决公司业务 90% 以上的离线数据分析任务。而对于需要频繁进行网络通信、内存消耗高、计算要求快速迭代的算法任务，MPI 无疑是最佳选择。MPI 是一种基于消息传递的并行计算框架，由于没有 IO 操作，性能优于 MapReduce。因此，阿里巴巴的算法平台选用 MPI 作为基础计算框架，其核心机器学习算法的开发都是基于阿里云 MaxCompute 的 MPI 实现的。

MaxCompute MPI 的处理流程如图 7.1 所示，与分布式计算系统的原理类似，不再赘述。其中伏羲为阿里云飞天系统的分布式调度系统，女娲为阿里云飞天系统的分布式一致性协同服务系统，盘古为阿里云飞天系统的分布式文件存储系统。

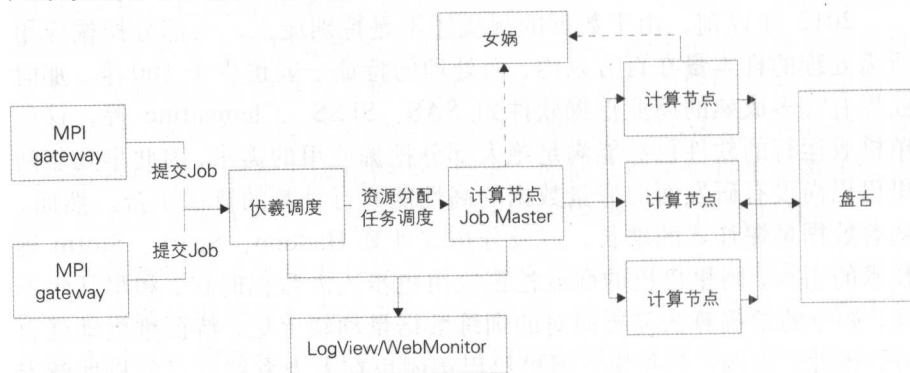


图 7.1 MaxCompute MPI 处理流程图

基于 MaxCompute MPI，目前阿里巴巴的算法平台已经集成了绝大部分业界主流的机器学习算法（见表 7.1），从传统的分类、聚类算法，到互联网应用中非常流行的协同过滤、PageRank 算法，再到当前最火热的深度学习算法，这些算法基本可以满足企业级数据挖掘应用的需要。配合阿里巴巴的大数据计算平台 MaxCompute，工程师们可以通过简易的命令式调用或拖拽式的可视化界面操作，将这些算法应用于自己的实际业务当中，体验机器学习算法与大数据结合后的强大功能与魅力。

表 7.1 基于 MaxCompute MPI 的机器学习算法

分 类	具体算法
分类算法	LogisticRegression、kNN、GBDT、DTC5.0、RandomForest、linearSVM、nonlinearSVM、NavieBayes、Bayes、Fisher 判别、马氏距离判别、标签传播分裂等
回归算法	LinearRegression、GBDT、LASSO、RidgeRegression、Factorization Machines、XGBoost 等
聚类算法	K-Means、Canopy、PSC 谱聚类、标签传播聚类、EM 聚类等
推荐算法	etrec 协同过滤、SVD 协同过滤、ALS 协同过滤等
深度学习	Word2Vec、Doc2Vec、CNN、DBN、DeepMatchModel 等
其他	PageRank、LDA、pLSA、关联规则、NMF、CRF、SVD、RankSVM、PCA、kcore、sssp、Modularity 计算等

注：etrec 是阿里巴巴集团搜索算法团队开发的运行于 MaxCompute 上的基于商品的协同过滤算法。

## 7.3 数据挖掘中台体系

在阿里巴巴集团，由于业务场景与商业智能分析需求的多样化，多个部门、多个商业智能及算法团队针对应用问题所提出的算法解决方案往往是独立的，通常一次数据挖掘的过程包括商业理解、数据准备、特征工程、模型训练、模型测试、模型部署、线上应用及效果反馈等环节。如果对于每个应用都完全独立地设计这么一套流程，那么对于阿里巴巴成千上万的挖掘应用而言无疑将产生巨大的时间与经济成本，带来大量

的重复建设和资源浪费。事实上，早在 2015 年，阿里巴巴集团便提出了中台战略，将一些通用的技术集成起来形成中台技术体系，为各业务部门提供统一、高效的技术服务，避免各业务部门在各自业务发展的过程中进行重复的技术建设造成不必要的资源浪费与时间消耗。对于数据挖掘技术而言，中台发展的思路同样适用，并且从长远来看，构建数据挖掘中台技术体系也是绝对有必要的。

就数据挖掘的商业场景而言，可以分为两大类应用：个体挖掘应用与关系挖掘应用。个体挖掘应用指对单个实体的行为特征进行预测与分析，如预测某商品的销量、划分某行业的价格区间等；关系挖掘应用指研究多个实体间的关系特征，如商品的相似关系、竞争关系等。就数据挖掘技术而言，其包含两大要素：数据和算法。数据是数据挖掘的起源与挖掘结果最终的承载形式，可以说任何数据挖掘的过程都是从数据里来，回数据里去，源于数据而高于数据；算法是数据挖掘的神经中枢，通过算法对原始数据进行加工，得到对业务更有价值的数据。因此，对于数据挖掘中台体系的设计也包含两大块：数据中台与算法中台；结合数据挖掘的商业场景，对这两大块的设计又分别从个体挖掘应用和关系挖掘应用两方面进行考虑。

### 7.3.1 挖掘数据中台

在数据挖掘的过程中包含两类数据：特征数据与结果数据。这两类数据很好理解，比如要预测某商品的销量，那么算法需要的特征变量其实就是特征数据，算法最终输出的商品销量的预测结果就是结果数据。对于特征数据，相信有一定数据挖掘工作经验的读者都知道，在挖掘项目中 80% 的时间可能都是在处理特征，这些特征的提取、清洗、标准化，以及基于业务场景的再组合和二次加工往往是我们工作内容的主体部分。试想，如果有一套标准、规范且索引方便的全局特征库，每个挖掘工程师只需访问几张物理表就能迅速地搜集到绝大部分自己想要的特征，是不是一件很酷的事情？同时，通过算法生成的结果数据也需要进行合理的分层存储。有的结果非常通用和基础化，可以在很多的业务场景中复用，有的结果则相对个性和场景化，只适用于某个具体的业务和

产品,因此需要对结果数据进行合理的分层,有效隔离通用性强和个性化强的结果,这样可以充分发挥通用性强的算法结果的作用,提升它的复用率,减少不必要的重复建设。

基于以上分析,我们把数据中台分为三层:特征层(Featural Data Mining Layer, FDM)、中间层和中间层(Application-oriented Data Mining Layer, ADM),其中中间层包括个体中间层(Individual Data Mining Layer, IDM)和关系中间层(Relational Data Mining Layer, RDM),如图7.2所示。

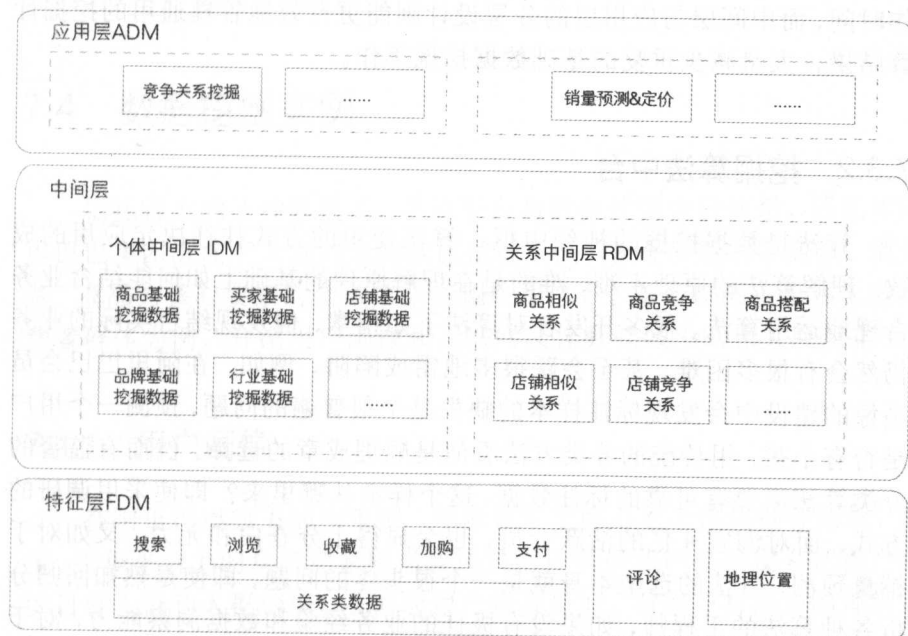


图 7.2 阿里巴巴数据挖掘中台

不同数据层的作用有所区别:

- FDM 层: 用于存储在模型训练前常用的特征指标, 并进行统一的清洗和去噪处理, 提升机器学习特征工程环节的效率。
- IDM 层: 个体挖掘指标中间层, 面向个体挖掘场景, 用于存储通用性强的结果数据, 主要包含商品、卖家、买家、行业等维度

的个体数据挖掘的相关指标。

- RDM 层：关系挖掘指标中间层，面向关系挖掘场景，用于存储通用性强的结果数据，主要包含商品间的相似关系、竞争关系，店铺间的相似关系、竞争关系等。
- ADM 层：用来沉淀比较个性偏应用的数据挖掘指标，比如用户偏好的类目、品牌等，这些数据已经过深度的加工处理，满足某一特点业务或产品的使用。

通过挖掘数据中台的建设，能够大幅度节省数据挖掘特征工程的工作时间，而中间层与应用层的分层设计则能更有效地管理通用的挖掘计算结果，大量减少重复的基础数据挖掘工作。

### 7.3.2 挖掘算法中台

算法是数据挖掘的神经中枢，算法使用的方式往往决定应用的成败。理解算法的原理不难，难的是在理解原理的基础上如何能结合业务合理地运用算法。很多开发者对算法了如指掌，但发现结合实际的业务仍然会有很多困难，甚至会踩很多地雷或陷阱。例如，在阿里巴巴会员画像的建设中会发现标注样本的缺失是个很普遍的问题，预测一个用户是否有小孩，用传统的分类方法看似是顺理成章的选择，然而有监督的分类算法需要有可靠的标注数据，这个样本从哪里来？即使采用调研的方式，面对淘宝 4 亿的活跃会员，也会显得十分苍白和无力。又如对于销量预测，方法的选型本身就是一个很头疼的问题，即使是熟知回归分析各种算法的工程师，如果没有极佳的业务经验和数据洞察能力，对于这个问题恐怕也没有太好的捷径找到最合适的方法。此时我们会想，如果能像金融领域的风控一样，有一套类似于评分卡建模的方法论和实操模板，那么处理业务问题的效率将会大大提升。而阿里巴巴数据挖掘算法中台建设的目的在于从各种各样的挖掘场景中抽象出有代表性的几类场景，并形成相应的方法论和实操模板。

按照个体挖掘应用和关系挖掘应用的分类方式，可以抽象出常见的几类数据挖掘应用场景——在个体挖掘应用中，消费者画像与业务指标预测是两类非常有代表性的场景；而在关系挖掘应用中，相似关系与竞



争关系是两类非常通用的关系挖掘应用,在此基础上构建的推荐系统与竞争分析系统,则是电商领域持续关注的两大热门话题。

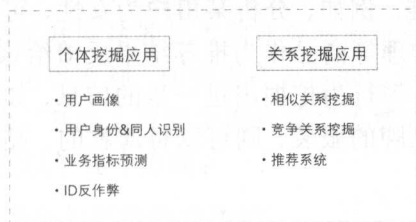


图 7.3 常见的数据挖掘应用

## 7.4 数据挖掘案例

依托强大的云计算技术、算法平台与数据挖掘中台体系,近几年阿里巴巴集团沉淀了大量的电商数据挖掘案例,并逐步形成以商家、消费者、商品为核心要素的全域数据挖掘应用体系。本节以淘宝市场上的消费者群体为例,介绍用户画像和互联网反作弊领域中的相关应用案例。

### 7.4.1 用户画像

在阿里巴巴旗下的淘宝网、虾米音乐上都不乏个性化推荐场景,淘宝、天猫平台上的众多商家则需要通过用户调研和产品研发来把握产品的目标人群和人群偏好,从而对用户投其所好。对用户有深刻的理解是网站推荐、企业经营制胜的重要一环。在传统企业中,获取用户的反馈信息耗时长、结果缺失,是个难关。然而,随着大数据热潮的兴起,快速捕捉海量用户行为并精确分析人群偏好等商业信息已经成为可能。作为个性化技术的重要基础,相比于传统企业的购物篮分析、问卷调查,在用户画像的塑造上具备技术的天然优势。

阿里全域数据提供了足够的数据基础,正是基于用户网购、搜索和娱乐影音等行为的数据洞察,可以利用数据分析辅以算法的视角对用户进行 360° 全方位的特征刻画。那么,究竟什么是用户画像?通俗地讲,



用户画像即是为用户打上各种各样的标签，如年龄、性别、职业、商品品牌偏好、商品类别偏好等。这些标签的数目越丰富，标签越细化，对用户的刻画就越精准。例如，分析某用户为女性，可能仅仅是将与女性相关的服装、个人护理等商品作为推荐结果反馈给该用户；但若根据用户以往的浏览、交易等行为挖掘出进一步的信息，如用户的地理信息为海南，买过某几类品牌的服装，则可以将薄款的、品牌风格相似的服装作为推荐结果。

一般而言，用户画像可以分为基础属性、购物偏好、社交关系、财富属性等几大类。对于刻画淘宝网购用户，则应侧重于他们在网购上的行为偏好。下面以用户女装风格偏好为例，讲解该用户标签是如何基于全域数据产出的。

购买过淘宝商品的读者对商品详情页都不会陌生，一件商品的关键特征除了反映在商品图片和详情页中以外，主要可以采集的信息是商品的标题以及参数描述。女装有哪些风格？首先需要将女装行业下的商品标题文本提取出来，对其进行分词，得到庞大的女装描绘词库。然而，淘宝商品的标题由卖家个人撰写，并不能保证其中的词语都与商品风格描述相关。因此，对于所得到的女装描绘词库，首先，需要根据词语权重去除无效的停用词，方法如计算 TF-IDF 值。其次，在女装商品的参数描述中，如果已经包含了一种商品风格，例如“通勤”“韩版”等常见风格，那么通过计算词库中词语与参数描述中风格词的相似度，可以过滤得到女装风格词库，利用无监督机器学习如 LDA 等方法可以计算出一种风格所包含的词汇及这些词汇的重要性。那么，买家偏好什么风格呢？在淘宝网上，买家拥有浏览、搜索、点击、收藏、加购物车以及交易等多种行为，针对每种行为赋予不同的行为强度（比如浏览行为强度弱于交易行为），再考虑该商品的风格元素组成，就能够通过合理的方式获知买家对该风格的偏好程度了。

对于这样的商品偏好计算，数据挖掘人员需要仔细分析用户偏好的商品的类型、品牌、风格元素、下单时间，这一系列行为可以构成复杂的行为模块。同理，利用机器学习算法，可以从用户行为中推测其身份，例如男生和女生、老年与青年偏好的商品和行为方式存在区别，根据一定的用户标记，最后能够预测出用户的基础身份信息。

### 7.4.2 互联网反作弊

在人们享受互联网带来的便捷和高效时,有一批人将其黑手伸向了这一领域,他们利用某些网站的技术和业务漏洞进行作弊,从而满足自己的灰色利益需求。可以看到,作弊黑产业链的滋生与发展使得人们的信息安全、资金安全,甚至人身安全面临着严重威胁。从业务上看,反作弊工作主要体现在以下几个方面:

#### (1) 账户/资金安全与网络欺诈防控

账户的安全性越来越重要,尤其是购物和理财的网站或 APP,其账户以及资金的安全更是维持用户信任的最后一道防线。账户隐私数据的泄露和非法交易问题不容忽视。

#### (2) 非人行为和账户识别

大量的非人行为和账户利用自动化程序来模拟人的注册、浏览、点击等行为,帮助一些商家达到提升商品排名或者攻击竞争对手的目的。

#### (3) 虚假订单与信用炒作识别

在平台类电商网站中,随着商家之间的竞争升级,虚假订单和信用炒作逐渐成为许多商家依赖的竞争手段。当下,虚假销量与 GMV 正在毒害健康的经营环境。

#### (4) 广告推广与 APP 安装反作弊

正当大量的企业开始投入人力和财力进行公司产品和品牌的宣传推广时,也有人通过自动化程序或者人工的方式产生虚假的点击和浏览行为,以此打击竞争对手,或者是从中获取直接的经济利益。同样在无线端,在 APP 以 CPA 为主要推广结算方式的情况下,大量的专业刷装机量的公司为很多 APP 产生虚假的装机量数据,以此达到各方获取不正当利益的目的。

#### (5) UGC 恶意信息检测

用户各类网站上产生的 UGC 信息,本可以帮助网站更好地了解用户的使用习惯以及潜在需求,帮助优化网站或者产品。但是,很多人却利用文本、图片、声音、视频等内容的复杂性和多样性,恶意传播色

情、诈骗、谣言、暴力等不正当信息，给互联网环境造成了很大的威胁。

随着作弊场景和手法的不断变化，各个维度的安全技术也在不断升级保护正常用户免受黑色利益链条的侵害，其中包括物理安全、网络安全、应用安全、数据安全等方面。在数据安全的保障工作中数据挖掘算法也扮演着十分重要的作用。从所采用的算法技术上说，反作弊方法主要包括如下几类：

### (1) 基于业务规则的方法

这类方法主要是根据实际的业务场景，不断地发现总结作弊和获利手法，通过反作弊规则的不断拓展或产品设计的完善来识别、缓解甚至消除作弊现象。比如在电商产品的搜索排名中，对刷单的销量采取类似于降权的处理方式，避免市场秩序受到影响，以此减少商家的刷单行为带来的不正当利益。或者是在 APP 安装反作弊工作中，通过查看单个设备的单日出出现城市数、登录账号数、设备 id 合法性等建立规则来衡量作弊情况。

这类方法的优点是精度高、可解释性强，能准确识别老的作弊方式；缺点是人力成本高，而且对新的作弊手法滞后性较强。

### (2) 基于有监督学习的方法

将基于有监督学习的方法应用于反作弊工作中，其基本思路是按照有监督分类算法的流程来建模，通过正负样本标记、特征提取、模型训练及预测等过程来识别作弊行为。比如在账户反欺诈场景下，以账户的属性信息和行为数据作为模型特征输入，以历史的欺诈事件作为标记样本，通过训练分类模型对当前账户的作弊风险进行预测评估。

但在反作弊领域内，此类方法遇到的最大问题是类不平衡现象。因为绝大多数用户及行为都是正常的，只有少数一些用户及行为是恶意的。比如相对于正常用户的转账行为，资金欺诈行为数量是极少的，但是这些极少的行为可能给用户带来无法挽回的经济损失。为了缓解这一问题，我们会在采样或者模型训练过程中进行一些技术处理，以减少类不平衡给识别结果造成的影响。

这类方法的优点是通用性强，人力成本主要集中在样本的标记和特

征的处理上；缺点是有些算法结果的可解释性不强，容易造成错判，需要辅以其他指标和方法进行综合判断。

### (3) 基于无监督学习的方法

在此类方法中较常见的是异常检测算法，该方法假设作弊行为极其罕见且在某些特征维度下和正常行为能够明显地区分开来。所以，假设检验、统计分析、聚类分析等手段常被用来做异常检测。比如我们发现账户的网站访问时间段分布有一定的规律，和人们日常的作息时间具有相关性，如果某个账户长期在凌晨发生大量的访问行为且转化率较低，那么就需要适当提高对应账户的风险等级。可以采用类似于上述算法，然后辅以一定的业务知识来综合判断行为的风险情况。

此类方法的优点是不需要标记正负样本，而且检测到的异常行为还可以沉淀到规则系统中；缺点是特征设计和提取的工作量大，需要在所有可能的风险维度下刻画行为特征。

除了上述方法外，类似于多媒体数据处理、图计算模型等方法也逐渐被用来处理反作弊问题。

此外，在实际应用中，上述几种方法并不是完全割裂的，有可能一个完整的反作弊系统会同时使用所有方法。而且，除了算法理论方面的工作外，在算法实现方面我们还会遇到很多问题，因此算法的实际应用工作也是十分重要的。这部分工作主要分为以下两个方面：

#### (1) 离线反作弊系统

离线反作弊系统主要包含规则判断、分类识别、异常检测等模块，通过历史行为和业务规则的沉淀，来判断未来行为的作弊情况。其优点是准确率较高，所使用的历史数据越多，判断结果越准确；缺点是时效性较差，无法及时给出判断结果。

#### (2) 实时反作弊系统

随着在某些场景下对时效性要求的不断提高，人们逐渐发现实时反作弊系统的必要性和重要性。所以，将离线中的许多规则和算法进行总结，在基本满足准确率和覆盖率的前提下抽取出其中计算速度较快的部分，以此来满足对实时性的要求。但是要求高的实时性可能要以一定的

准确率为代价，而且由于数据需要进行实时采集和计算，所以对数据存储和计算系统的性能要求也非常高。

通过对现有的作弊以及反作弊相关内容的介绍，我们可以看到，这一领域的很多问题暂时未得到解决，未来还面临着诸多挑战。比如：

### (1) 作弊手段的多样性和多变性

随着黑产公司的规模化和“正规”化，作弊与获利手段的多样性和多变性越来越明显，而反作弊系统如何能更及时地发现识别出新的手法和灰色利益链条，这是挖掘算法所面临的一个重要挑战。

### (2) 算法的及时性和准确性

由于普通用户的隐私和安全意识越来越强，所以反作弊系统的准确性和及时性要求越来越高，不仅要尽可能减少误判的情况，还需要及时发现真正的作弊行为，在给用户造成更大的损失之前对其实施有效的控制措施。

### (3) 数据及作弊手段的沉淀和逆向反馈

随着反作弊系统的升级改造，算法工程师无论在业务还是算法上都积累了相当多的数据和经验，如何将这些作弊手法以及反作弊手段进行通用性的沉淀，以及高效地逆向反馈到新的反作弊系统中，保证算法能紧跟市场脚步，也是反作弊工程师需要考虑的重要问题。

大数据时代为人们带来了丰富的基础数据和应用方式，也对信息安全提出了更高的要求，相信数据挖掘领域的不断发展能为这一方面的工作带来创新和突破。

## 第2篇

# 数据模型篇

- 第8章 大数据领域建模综述
- 第9章 阿里巴巴数据整合及管理体系
- 第10章 维度设计
- 第11章 事实表设计

## 第8章

# 大数据领域 建模综述

### 8.1 为什么需要数据建模

随着 DT 时代互联网、智能设备及其他信息技术的发展，数据爆发式增长，如何将这些数据进行有序、有结构地分类组织和存储是我们面临的一个挑战。

如果把数据看作图书馆里的书，我们希望看到它们在书架上分门别类地放置；如果把数据看作城市的建筑，我们希望城市规划布局合理；如果把数据看作电脑文件和文件夹，我们希望按照自己的习惯有很好的文件夹组织方式，而不是糟糕混乱的桌面，经常为找一个文件而不知所措。

数据模型就是数据组织和存储方法，它强调从业务、数据存取和使

用角度合理存储数据。Linux 的创始人 Torvalds 有一段关于“什么才是优秀程序员”的话：“烂程序员关心的是代码，好程序员关心的是数据结构和它们之间的关系”，其阐述了数据模型的重要性。有了适合业务和基础数据存储环境的模型，那么大数据就能获得以下好处。

- 性能：良好的数据模型能帮助我们快速查询所需要的数据，减少数据的 I/O 吞吐。
- 成本：良好的数据模型能极大地减少不必要的冗余数据，也能实现计算结果复用，极大地降低大数据系统中的存储和计算成本。
- 效率：良好的数据模型能极大地改善用户使用数据的体验，提高使用数据的效率。
- 质量：良好的数据模型能改善数据统计口径的不一致性，减少数据计算错误的可能性。

因此，毋庸置疑，大数据系统需要数据模型方法来帮助更好地组织和存储数据，以便在性能、成本、效率和质量之间取得最佳平衡。

## 8.2 关系数据库系统和数据仓库

E.F.Codd 是关系数据库的鼻祖，他首次提出了数据库系统的关系模型，开创了数据库关系方法和关系数据理论的研究。随着一大批大型关系数据库商业软件（如 Oracle、Informix、DB2 等）的兴起，现代企业信息系统几乎都使用关系数据库来存储、加工和处理数据。数据仓库系统也不例外，大量的数据仓库系统依托强大的关系数据库能力存储和处理数据，其采用的数据模型方法也是基于关系数据库理论的。虽然近年来大数据的存储和计算基础设施在分布式方面有了飞速的发展，NoSQL 技术也曾流行一时，但是不管是 Hadoop、Spark 还是阿里巴巴集团的 MaxCompute 系统，仍然在大规模使用 SQL 进行数据的加工和处理，仍然在用 Table 存储数据，仍然在使用关系理论描述数据之间的关系，只是在大数据领域，基于其数据存取的特点在关系数据模型的范式上有了不同的选择而已。关于范式的详细说明和定义，以及其他一些



关系数据库的理论是大数据领域建模的基础,有兴趣的读者可以参考相关的经典数据库理论书籍,如《数据库系统概念》。

## 8.3 从 OLTP 和 OLAP 系统的区别看模型方法论的选择

OLTP 系统通常面向的主要数据操作是随机读写,主要采用满足 3NF 的实体关系模型存储数据,从而在事务处理中解决数据的冗余和一致性问题;而 OLAP 系统面向的主要数据操作是批量读写,事务处理中的一致性不是 OLAP 所关注的,其主要关注数据的整合,以及在一次性的复杂大数据查询和处理中的性能,因此它需要采用一些不同的数据建模方法。

## 8.4 典型的数据仓库建模方法论

### 8.4.1 ER 模型

数据仓库之父 Bill Inmon 提出的建模方法是从全企业的高度设计一个 3NF 模型,用实体关系 (Entity Relationship, ER) 模型描述企业业务,在范式理论上符合 3NF。数据仓库中的 3NF 与 OLTP 系统中的 3NF 的区别在于,它是站在企业角度面向主题的抽象,而不是针对某个具体业务流程的实体对象关系的抽象。其具有以下几个特点:

- 需要全面了解企业业务和数据。
- 实施周期非常长。
- 对建模人员的能力要求非常高。

采用 ER 模型建设数据仓库模型的出发点是整合数据,将各个系统

中的数据以整个企业角度按主题进行相似性组合和合并,并进行一致性处理,为数据分析决策服务,但是并不能直接用于分析决策。

其建模步骤分为三个阶段。

- 高层模型:一个高度抽象的模型,描述主要的主题以及主题间的关系,用于描述企业的业务总体概况。
- 中层模型:在高层模型的基础上,细化主题的数据项。
- 物理模型(也叫底层模型):在中层模型的基础上,考虑物理存储,同时基于性能和平台特点进行物理属性的设计,也可能做一些表的合并、分区的设计等。

ER模型在实践中最典型的代表是Teradata公司基于金融业务发布的FS-LDM(Financial Services Logical Data Model),它通过对金融业务的高度抽象和总结,将金融业务划分为10大主题,并以设计面向金融仓库模型的核心为基础,企业基于此模型做适当调整和扩展就能快速落地实施。

## 8.4.2 维度模型

维度模型是数据仓库领域的Ralph Kimball大师所倡导的,他的*The Data Warehouse Toolkit-The Complete Guide to Dimensional Modeling*是数据仓库工程领域最流行的数据仓库建模的经典。

维度建模从分析决策的需求出发构建模型,为分析需求服务,因此它重点关注用户如何更快速地完成需求分析,同时具有较好的大规模复杂查询的响应性能。其典型的代表是星形模型,以及在一些特殊场景下使用的雪花模型。其设计分为以下几个步骤。

- 选择需要进行分析决策的业务过程。业务过程可以是单个业务事件,比如交易的支付、退款等;也可以是某个事件的状态,比如当前的账户余额等;还可以是一系列相关业务事件组成的业务流程,具体需要看我们分析的是某些事件发生情况,还是当前状态,或是事件流转效率。
- 选择粒度。在事件分析中,我们要预判所有分析需要细分的程度,

从而决定选择的粒度。粒度是维度的一个组合。

- 识别维表。选择好粒度之后，就需要基于此粒度设计维表，包括维度属性，用于分析时进行分组和筛选。
- 选择事实。确定分析需要衡量的指标。

### 8.4.3 Data Vault 模型

Data Vault 是 Dan Linstedt 发起创建的一种模型，它是 ER 模型的衍生，其设计的出发点也是为了实现数据的整合，但不能直接用于数据分析决策。它强调建立一个可审计的基础数据层，也就是强调数据的历史性、可追溯性和原子性，而不要求对数据进行过度的一致性处理和整合；同时它基于主题概念将企业数据进行结构化组织，并引入了更进一步的范式处理来优化模型，以应对源系统变更的扩展性。Data Vault 模型由以下几部分组成。

- Hub：是企业的核心业务实体，由实体 key、数据仓库序列代理键、装载时间、数据来源组成。
- Link：代表 Hub 之间的关系。这里与 ER 模型最大的区别是将关系作为一个独立的单元抽象，可以提升模型的扩展性。它可以直接描述 1:1、1:n 和 n:n 的关系，而不需要做任何变更。它由 Hub 的代理键、装载时间、数据来源组成。
- Satellite：是 Hub 的详细描述内容，一个 Hub 可以有多个 Satellite。它由 Hub 的代理键、装载时间、来源类型、详细的 Hub 描述信息组成。

Data Vault 模型比 ER 模型更容易设计和产出，它的 ETL 加工可实现配置化。通过 Dan Linstedt 的比喻更能理解 Data Vault 的核心思想：Hub 可以想象成人的骨架，那么 Link 就是连接骨架的韧带，而 Satellite 就是骨架上面的血肉。看如下实例（来自 *Data Vault Modeling Guide*，作者 Hans Hultgren），如图 8.1 所示。

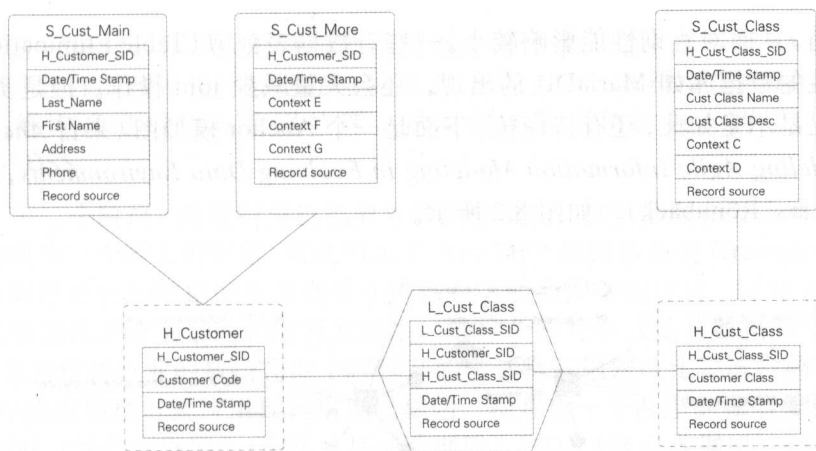


图 8.1 Data Vault 模型实例

#### 8.4.4 Anchor 模型

Anchor 对 Data Vault 模型做了进一步规范化处理，Lars. Rönneback 的初衷是设计一个高度可扩展的模型，其核心思想是所有的扩展只是添加而不是修改，因此将模型规范到 6NF，基本变成了 k-v 结构化模型。我们看一下 Anchor 模型的组成。

- Anchors：类似于 Data Vault 的 Hub，代表业务实体，且只有主键。
- Attributes：功能类似于 Data Vault 的 Satellite，但是它更加规范化，将其全部 k-v 结构化，一个表只有一个 Anchors 的属性描述。
- Ties：就是 Anchors 之间的关系，单独用表来描述，类似于 Data Vault 的 Link，可以提升整体模型关系的扩展能力。
- Knots：代表那些可能会在多个 Anchors 中公用的属性的提炼，比如性别、状态等这种枚举类型且被公用的属性。

在上述四个基本对象的基础上，又可以细划分为历史的和非历史的，其中历史的会以时间戳加多条记录的方式记录数据的变迁历史。

Anchor 模型的创建者以此方式来获取极大的可扩展性，但是也会增加非常多的查询 join 操作。创建者的观点是，数据仓库中的分析查询只是基于一小部分字段进行的，类似于列存储结构，可以大大减少数据

扫描, 从而对查询性能影响较小。一些有数据表裁剪 (Table Elimination) 特性的数据库如 MariaDB 的出现, 还会大量减少 join 操作。但是实际情况是不是如此, 还有待商榷。下面是一个 Anchor 模型图 (来自 *Anchor Modeling-Agile Information Modeling in Evolving Data Environments*, 作者 Lars. Rönnbäck), 如图 8.2 所示。

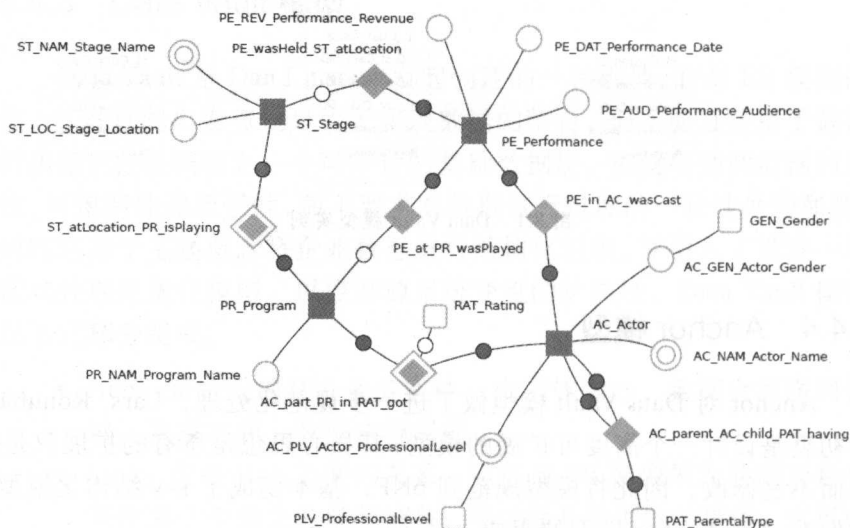


图 8.2 Anchor 模型图

## 8.5 阿里巴巴数据模型实践综述

阿里巴巴集团很早就已经把大数据作为其战略目标实施, 而且其各个业务也非常依赖数据支撑运营, 那么阿里巴巴究竟采取何种方法构建自己的数据仓库模型呢? 阿里巴巴的数据仓库模型建设经历了多个发展阶段。

第一个阶段: 完全应用驱动的时代, 阿里巴巴的第一代数据仓库系统构建在 Oracle 上, 数据完全以满足报表需求为目的, 将数据以与源结构相同的方式同步到 Oracle (称作 ODS 层), 数据工程师基于 ODS

数据进行统计,基本没有系统化的模型方法体系,完全基于对 Oracle 数据库特性的利用进行数据存储和加工,部分采用一些维度建模的缓慢变化维方式进行历史数据处理。这时候的数据架构只有两层,即 ODS+DSS。

第二个阶段:随着阿里巴巴业务的快速发展,数据量也在飞速增长,性能成为一个较大的问题,因此引入了当时 MPP 架构体系的 Greenplum,同时阿里巴巴的数据团队也在着手进行一定的数据架构优化,希望通过一些模型技术改变烟囱式的开发模型,消除一些冗余,提升数据的一致性。来自传统行业的数据仓库工程师开始尝试将工程领域比较流行的 ER 模型+维度模型方式应用到阿里巴巴集团,构建出一个四层的模型架构,即 ODL(操作数据层)+BDL(基础数据层)+IDL(接口数据层)+ADL(应用数据层)。ODL 和源系统保持一致;BDL 希望引入 ER 模型,加强数据的整合,构建一致的基础数据模型;IDL 基于维度模型方法构建集市层;ADL 完成应用的个性化和基于展现需求的数据组装。在此期间,我们在构建 ER 模型时遇到了比较大的困难和挑战,互联网业务的快速发展、人员的快速变化、业务知识功底的不够全面,导致 ER 模型设计迟迟不能产出。至此,我们也得到了一个经验:在不太成熟、快速变化的业务面前,构建 ER 模型的风险非常大,不太适合去构建 ER 模型。

第三个阶段:阿里巴巴集团的业务和数据还在飞速发展,这时候迎来了以 Hadoop 为代表的分布式存储计算平台的快速发展,同时阿里巴巴集团自主研发的分布式计算平台 MaxCompute 也在紧锣密鼓地进行着。我们在拥抱分布式计算平台的同时,也开始建设自己的第三代模型架构,这时候需要找到既适合阿里巴巴集团业务发展,又能充分利用分布式计算平台能力的模型方式。我们选择了以 Kimball 的维度建模为核心理念的模型方法论,同时对其进行了一定的升级和扩展,构建了阿里巴巴集团的公共层模型数据架构体系。

数据公共层建设的目的是着力解决数据存储和计算的共享问题。阿里巴巴集团当下已经发展为多个 BU,各个业务产生庞大的数据,并且数据每年以近 2.5 倍的速度在增长,数据的增长远远超过业务的增长,带来的成本开销也是非常令人担忧的。

阿里巴巴数据公共层建设的指导方法是一套统一化的集团数据整合及管理的方法体系(在内部这一体系称为“OneData”),其包括一致性的指标定义体系、模型设计方法体系以及配套工具。

## 第9章

# 阿里巴巴数据整合及管理体系

面对爆炸式增长的数据，如何建设高效的数据模型和体系，对这些数据进行有序和有结构地分类组织和存储，避免重复建设和数据不一致性，保证数据的规范性，一直是大数据系统建设不断追求的方向。

OneData 即是阿里巴巴内部进行数据整合及管理的方法体系和工具。阿里巴巴的大数据工程师在这一体系下，构建统一、规范、可共享的全域数据体系，避免数据的冗余和重复建设，规避数据烟囱和不一致性，充分发挥阿里巴巴在大数据海量、多样性方面的独特优势。借助这一统一化数据整合及管理的方法体系，我们构建了阿里巴巴的数据公共层，并可以帮助相似的大数据项目快速落地实现。下面重点介绍 OneData 体系和实施方法论。

### 9.1 概述

阿里巴巴集团大数据建设方法论的核心是：从业务架构设计到模型

设计,从数据研发到数据服务,做到数据可管理、可追溯、可规避重复建设。目前,阿里巴巴集团数据公共层团队已把这套方法论沉淀为产品,以帮助数据 PD、数据模型师和 ETL 工程师建设阿里的大数据。这一体系包含方法论以及相关产品。

### 9.1.1 定位及价值

建设统一的、规范化的数据接入层(ODS)和数据中间层(DWD和DWS),通过数据服务和数据产品,完成服务于阿里巴巴的大数据系统建设,即数据公共层建设。提供标准化的(Standard)、共享的(Shared)、数据服务(Service)能力,降低数据互通成本,释放计算、存储、人力等资源,以消除业务和技术之痛。

### 9.1.2 体系架构

体系架构图如图 9.1 所示。

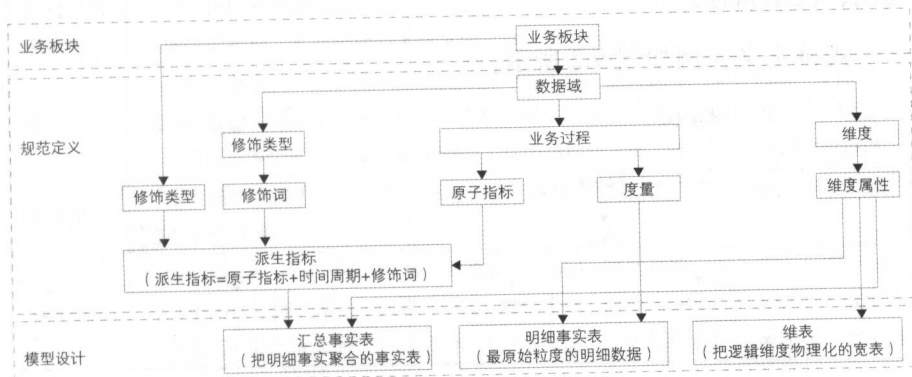


图 9.1 体系架构图

**业务板块：**由于阿里巴巴集团业务生态庞大,所以根据业务的属性划分出几个相对独立的业务板块,业务板块之间的指标或业务重叠性较



小。如电商业务板块涵盖淘系、B2B 系和 AliExpress 系等。

**规范定义：**阿里数据业务庞大，结合行业的数据仓库建设经验和阿里数据自身特点，设计出的一套数据规范命名体系，规范定义将会被用在模型设计中。后面章节将会详细说明。

**模型设计：**以维度建模理论为基础，基于维度建模总线架构，构建一致性的维度和事实（进行规范定义）。同时，在落地表模型时，基于阿里自身业务特点，设计出一套表规范命名体系。后面章节将会详细说明。

## 9.2 规范定义

规范定义指以维度建模作为理论基础，构建总线矩阵，划分和定义数据域、业务过程、维度、度量 / 原子指标、修饰类型、修饰词、时间周期、派生指标。

规范定义实例如图 9.2 所示。

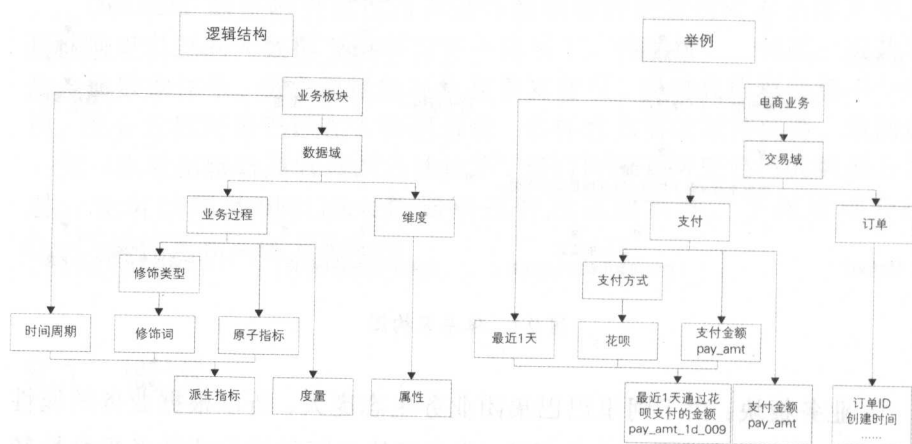


图 9.2 规范定义实例

## 9.2.1 名词术语

名词术语解释如表 9.1 所示。

表 9.1 名词术语解释

名词术语	解 释
数据域	指面向业务分析,将业务过程或者维度进行抽象的集合。其中,业务过程可以概括为一个个不可拆分的行为事件,在业务过程之下,可以定义指标;维度是指度量的环境,如买家下单事件,买家是维度。为保障整个体系的生命力,数据域是需要抽象提炼,并且长期维护和更新的,但不轻易变动。在划分数据域时,既能涵盖当前所有的业务需求,又能在新业务进入时无影响地被包含进已有的数据域中和扩展新的数据域
业务过程	指企业的业务活动事件,如下单、支付、退款都是业务过程。请注意,业务过程是一个不可拆分的行为事件,通俗地讲,业务过程就是企业活动中的事件
时间周期	用来明确数据统计的时间范围或者时间点,如最近 30 天、自然周、截至当日等
修饰类型	是对修饰词的一种抽象划分。修饰类型从属于某个业务域,如日志域的访问终端类型涵盖无线端、PC 端等修饰词
修饰词	指除了统计维度以外指标的业务场景限定抽象。修饰词隶属于一种修饰类型,如在日志域的访问终端类型下,有修饰词 PC 端、无线端等
度量 / 原子指标	原子指标和度量含义相同,基于某一业务事件行为下的度量,是业务定义中不可再拆分的指标,具有明确业务含义的名词,如支付金额
维度	维度是度量的环境,用来反映业务的一类属性,这类属性的集合构成一个维度,也可以称为实体对象。维度属于一个数据域,如地理维度(其中包括国家、地区、省以及城市等级别的内容)、时间维度(其中包括年、季、月、周、日等级别的内容)
维度属性	维度属性隶属于一个维度,如地理维度里面的国家名称、国家 ID、省份名称等都属于维度属性
派生指标	派生指标=一个原子指标+多个修饰词(可选)+时间周期。可以理解为对原子指标业务统计范围的圈定。如原子指标:支付金额,最近 1 天海外买家支付金额则为派生指标(最近 1 天为时间周期,海外为修饰词,买家作为维度,而不作为修饰词)

## 9.2.2 指标体系

本文在讲述指标时,会涵盖其组成体系(原子指标、派生指标、修饰类型、修饰词、时间周期),将它们作为一个整体来解读。

## 1. 基本原则

### (1) 组成体系之间的关系

- 派生指标由原子指标、时间周期修饰词、若干其他修饰词组合得到（见图 9.3）。



图 9.3 派生指标

- 原子指标、修饰类型及修饰词，直接归属在业务过程下，其中修饰词继承修饰类型的数据域。
- 派生指标可以选择多个修饰词，修饰词之间的关系为“或”或者“且”，由具体的派生指标语义决定。
- 派生指标唯一归属一个原子指标，继承原子指标的数据域，与修饰词的数据域无关。

一般而言，事务型指标和存量型指标（见下文定义）只会唯一定位到一个业务过程，如果遇到同时有两个行为发生、需要多个修饰词、生成一个派生指标的情况，则选择时间靠后的行为创建原子指标，选择时间靠前的行为创建修饰词。

- 原子指标有确定的英文字段名、数据类型和算法说明；派生指标要继承原子指标的英文名、数据类型和算法要求。

### (2) 命名约定

- 命名所用术语。指标命名，尽量使用英文简写，其次是英文，当指标英文名太长时，可考虑用汉语拼音首字母命名。如中国质造，用 zgzc。在 OneData 工具中维护着常用的名词术语，以用来进行命名。
- 业务过程。英文名：用英文或英文的缩写或者中文拼音简写；中文名：具体的业务过程中文即可，如图 9.4 所示。

业务过程详情	
业务板块	电商业务
数据域	互动域
中文名称	内营订阅关注
英文名称	sns_follow
描述	对人进行关注 (e.g. 淘宝达人粉丝对达人进行关注); 对内容集合进行订阅 (e.g. 淘宝头条元宝妈妈、手淘社区的护肤大将) 注意:
状态: 审核通过	

图 9.4 业务过程命名示例

关于存量型指标 (见下文定义) 对应的业务过程的约定: 实体对象英文名+\_stock。如在线会员数、一星会员数等, 其对应的业务过程为 mbr\_stock; 在线商品数、商品 SKU 种类小于 5 的商品数, 其对应的业务过程为 itm\_stock。

- 原子指标。英文名: 动作+度量; 中文名: 动作+度量。原子指标必须挂靠在某业务过程下, 如图 9.5 所示。

原子指标详情	
状态: 审核通过	
<b>基本属性</b>	
中文名称	支付金额
英文名称	pay_order_amt
数据类型	DOUBLE
单位	空
描述信息	线上支付金额, 拍卖模式为订单金额最大金额为订单支付金额, 且线下支付
<b>归属信息</b>	
业务板块	电商业务
数据域	交易域
业务过程	订单支付

图 9.5 原子指标详情

- 修饰词。只有时间周期才会有英文名，且长度为 2 位，加上 “\_” 为 3 位，例如 1d。其他修饰词无英文名。

阿里巴巴常用的时间周期修饰词如表 9.2 所示。

表 9.2 阿里巴巴常用的时间周期修饰词

中 文 名	英 文 名	中 文 名	英 文 名
最近 1 天	1d	自然月	cm
最近 3 天	3d	自然季度	cq
最近 7 天	1w	截至当日	td
最近 14 天	2w	年初截至当日	sd
最近 30 天	1m	零点截至当前	tt
最近 60 天	2m	财年	fy
最近 90 天	3m	最近 1 小时	1h
最近 180 天	6m	准实时	ts
180 天以前	bh	未来 7 天	f1w
自然周	cw	未来 4 周	f4w

- 派生指标。英文名：原子指标英文名+时间周期修饰词（3 位，例如 1d）+序号（4 位，例如 001）；中文名：时间周期修饰词+[其他修饰词]+原子指标。

在 OneData 工具中，英文名与中文名都会由 OneData 工具自动生成，如图 9.6 所示。

图 9.6 派生指标命名示例

为了控制派生指标的英文名称过长,在英文名的理解和规范上做了取舍,所有修饰词的含义都纳入了序号中。序号是根据原子指标+派生指标自增的。

### (3) 算法

原子指标、修饰词、派生指标的算法说明必须让各种使用人员看得明白,包括:

- 算法概述——算法对应的用户容易理解的阐述。
- 举例——通过具体例子帮助理解指标算法。
- SQL 算法说明——对于派生指标给出 SQL 的写法或者伪代码。

## 2. 操作细则

### (1) 派生指标的种类

派生指标可以分为三类:事务型指标、存量型指标和复合型指标。按照其特性不同,有些必须新建原子指标,有些可以在其他类型原子指标的基础上增加修饰词形成派生指标。

- 事务型指标:是指对业务活动进行衡量的指标。例如新发商品数、重发商品数、新增注册会员数、订单支付金额,这类指标需维护原子指标及修饰词,在此基础上创建派生指标。
- 存量型指标:是指对实体对象(如商品、会员)某些状态的统计。例如商品总数、注册会员总数,这类指标需维护原子指标及修饰词,在此基础上创建派生指标,对应的时间周期一般为“历史截至当前某个时间”。
- 复合型指标:是在事务型指标和存量型指标的基础上复合而成的。例如浏览 UV-下单买家数转化率,有些需要创建新原子指标,有些则可以在事务型或存量型原子指标的基础上增加修饰词得到派生指标。

### (2) 复合型指标的规则

- 比率型:创建原子指标,如 CTR、浏览 UV-下单买家数转化率、满意率等。例如,“最近 1 天店铺首页 CTR”,原子指标为“CTR”,

时间周期为“最近 1 天”，修饰类型为“页面类型”，修饰词为“店铺首页”。

- 比例型：创建原子指标，如百分比、占比。例如“最近 1 天无线支付金额占比”，原子指标为“支付金额占比”，修饰类型为“终端类型”，修饰词为“无线”。
- 变化量型：不创建原子指标，增加修饰词，在此基础上创建派生指标。例如，“最近 1 天订单支付金额上 1 天变化量”，原子指标为“订单支付金额”，时间周期为“最近 1 天”，修饰类型为“统计方法”，修饰词为“上 1 天变化量”。
- 变化率型：创建原子指标。例如，“最近 7 天海外买家支付金额上 7 天变化率”，原子指标为“支付金额变化率”，修饰类型为“买家地域”，修饰词为“海外买家”。
- 统计型（均值、分位数等）：不创建原子指标，增加修饰词，在此基础上创建派生指标；在修饰类型“统计方法”下增加修饰词，如人均、日均、行业平均、商品平均、90 分位数、70 分位数等。例如，“自然月日均 UV”，原子指标为“UV”，修饰类型为“统计方法”，修饰词为“日均”。
- 排名型：创建原子指标，一般为 top\_xxx\_xxx，有时会同时选择 rank 和 top\_xxx\_xxx 组合使用。创建派生指标时选择对应的修饰词如下：
  - 统计方法（如降序、升序）。
  - 排名名次（如 TOP10）。
  - 排名范围（如行业、省份、一级来源等）。
  - 根据什么排序（如搜索次数、PV）。

示例如图 9.7 所示。

- 对象集合型：主要是指数据产品和应用需要展现数据时，将一些对象以 k-v 对的方式存储在一个字段中，方便前端展现。比如趋势图、TOP 排名对象等。其定义方式是，创建原子指标，一般为 xxx 串；创建派生指标时选择对应的修饰词如下：

- ▶ 统计方法（如降序、升序）。
- ▶ 排名名次（如 TOP10）。
- ▶ 排名范围（如行业、区域）。

图 9.7 派生指标示例 1

示例如图 9.8 所示。

图 9.8 派生指标示例 2

### 3. 其他规则

#### (1) 上下层级派生指标同时存在时

如最近 1 天支付金额和最近 1 天 PC 端支付金额，建议使用前者，把 PC 端作为维度属性存放在物理表中体现。



## (2) 父子关系原子指标存在时

当父子关系原子指标存在时,派生指标使用子原子指标创建派生指标。如 PV、IPV (商品详情页 PV),当统计商品详情页 PV 时,优先选择子原子指标。

## 9.3 模型设计

### 9.3.1 指导理论

阿里巴巴集团数据公共层设计理念遵循维度建模思想,可参考 *Star Schema-The Complete Reference* 和 *The Data Warehouse Toolkit-The Definitive Guide to Dimensional Modeling*。数据模型的维度设计主要以维度建模理论为基础,基于维度数据模型总线架构,构建一致性的维度和事实。

### 9.3.2 模型层次

阿里巴巴的数据团队把表数据模型分为三层:操作数据层(ODS)、公共维度模型层(CDM)和应用数据层(ADS),其中公共维度模型层包括明细数据层(DWD)和汇总数据层(DWS)。模型层次关系如图 9.9 所示。

**操作数据层(ODS):**把操作系统数据几乎无处理地存放在数据仓库系统中。

- 同步:结构化数据增量或全量同步到 MaxCompute。
- 结构化:非结构化(日志)结构化处理并存储到 MaxCompute。
- 累积历史、清洗:根据数据业务需求及稽核和审计要求保存历史数据、清洗数据。



图 9.9 模型层次关系图

**公共维度模型层 (CDM)：**存放明细事实数据、维表数据及公共指标汇总数据，其中明细事实数据、维表数据一般根据 ODS 层数据加工生成；公共指标汇总数据一般根据维表数据和明细事实数据加工生成。

CDM 层又细分为 DWD 层和 DWS 层，分别是明细数据层和汇总数据层，采用维度模型方法作为理论基础，更多地采用一些维度退化手法，将维度退化至事实表中，减少事实表和维表的关联，提高明细数据表的易用性；同时在汇总数据层，加强指标的维度退化，采取更多的宽表化手段构建公共指标数据层，提升公共指标的复用性，减少重复加工。其主要功能如下。

- 组合相关和相似数据：采用明细宽表，复用关联计算，减少数据扫描。
- 公共指标统一加工：基于 OneData 体系构建命名规范、口径一致和算法统一的统计指标，为上层数据产品、应用和服务提供公共指标；建立逻辑汇总宽表。
- 建立一致性维度：建立一致的数据分析维表，降低数据计算口径、算法不统一的风险。

**应用数据层 (ADS)：**存放数据产品个性化的统计指标数据，根据 CDM 层与 ODS 层加工生成。

- 个性化指标加工：不公用性、复杂性（指数型、比值型、排名型指标）。
- 基于应用的数据组装：大宽表集市、横表转纵表、趋势指标串。

其模型架构如图 9.10 所示。阿里巴巴通过构建全域的公共层数据，极大地控制了数据规模的增长趋势，同时在全体的数据研发效率、成本节约、性能改进方面都有不错的效果。

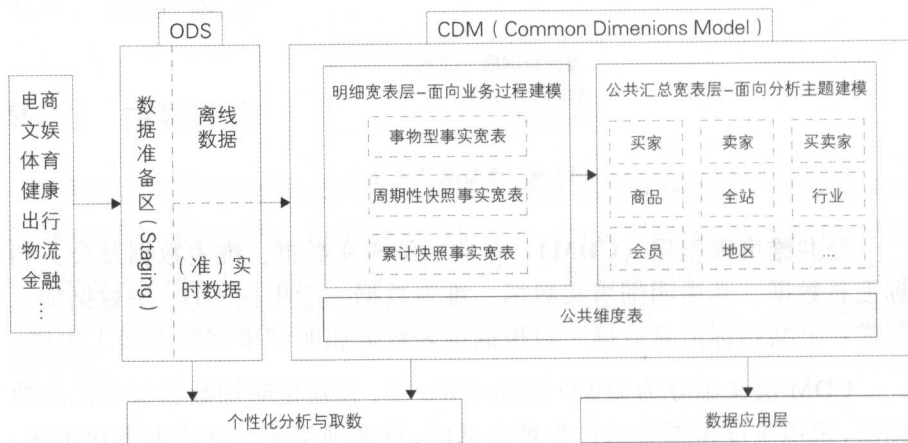


图 9.10 模型架构图

数据调用服务优先使用公共维度模型层（CDM）数据，当公共层没有数据时，需评估是否需要创建公共层数据，当不需要建设公用的公共层时，方可直接使用操作数据层（ODS）数据。应用数据层（ADS）作为产品特有的个性化数据一般不对外提供数据服务，但是 ADS 作为被服务方也需要遵守这个约定。

### 9.3.3 基本原则

#### 1. 高内聚和低耦合

一个逻辑或者物理模型由哪些记录和字段组成，应该遵循最基本的软件设计方法论的高内聚和低耦合原则。主要从数据业务特性和访问特

性两个角度来考虑：将业务相近或者相关、粒度相同的数据设计为一个逻辑或者物理模型；将高概率同时访问的数据放一起，将低概率同时访问的数据分开存储。

## 2. 核心模型与扩展模型分离

建立核心模型与扩展模型体系，核心模型包括的字段支持常用的核心业务，扩展模型包括的字段支持个性化或少量应用的需要，不能让扩展模型的字段过度侵入核心模型，以免破坏核心模型的架构简洁性与可维护性。

## 3. 公共处理逻辑下沉及单一

越是底层公用的处理逻辑越应该在数据调度依赖的底层进行封装与实现，不要让公用的处理逻辑暴露给应用层实现，不要让公共逻辑多处同时存在。

## 4. 成本与性能平衡

适当的数据冗余可换取查询和刷新性能，不宜过度冗余与数据复制。

## 5. 数据可回滚

处理逻辑不变，在不同时间多次运行数据结果确定不变。

## 6. 一致性

具有相同含义的字段在不同表中的命名必须相同，必须使用规范定义中的名称。

## 7. 命名清晰、可理解

表命名需清晰、一致，表名需易于消费者理解和使用。

## 9.4 模型实施

如何从具体的需求或项目转换为可实施的解决方案,如何进行需求分析、架构设计、详细模型设计等,则是模型实施过程中讨论的内容。本节先简单介绍业界常用的模型实施过程,然后重点讲解阿里巴巴 OneData 模型设计理论及实施过程。

### 9.4.1 业界常用的模型实施过程

#### 1. Kimball 模型实施过程

Kimball 维度建模主要探讨需求分析、高层模型、详细模型和模型审查整个过程。

构建维度模型一般要经历三个阶段:第一个阶段是高层设计时期,定义业务过程维度模型的范围,提供每种星形模式的技术和功能描述;第二个阶段是详细模型设计时期,对每个星形模型添加属性和度量信息;第三个阶段是进行模型的审查、再设计和验证等工作,第四个阶段是产生详细设计文档,提交 ETL 设计和开发。

##### (1) 高层模型

高层模型设计阶段的直接产出目标是创建高层维度模型图,它是对业务过程中的维表和事实表的图形描述。确定维表创建初始属性列表,为每个事实表创建提议度量。

##### (2) 详细模型

详细的维度建模过程是为高层模型填补缺失的信息,解决设计问题,并不断测试模型能否满足业务需求,确保模型的完备性。确定每个维表的属性和每个事实表的度量,并确定信息来源的位置、定义,确定属性和度量如何填入模型的初步业务规则。

##### (3) 模型审查、再设计和验证

本阶段主要召集相关人员进行模型的审查和验证,根据审查结果对详细维度进行再设计。

#### (4) 提交 ETL 设计和开发

最后,完成模型详细设计文档,提交 ETL 开发人员,进入 ETL 设计和开发阶段,由 ETL 人员完成物理模型的设计和开发。

上述内容主要引用自 Ralph Kimball 等的 *The Data Warehouse Lifecycle Toolkit*,具体细节请参考原著作。

### 2. Inmon 模型实施过程

Inmon 对数据模型的定位是:扮演着通往数据仓库其他部分的智能路线图的角色。由于数据仓库的建设不是一蹴而就的,为了协调不同人员的工作以及适应不同类型的用户,非常有必要建立一个路线图——数据模型,描述数据仓库各部分是如何结合在一起的。

Inmon 将模型划分为三个层次,分别是 ERD (Entity Relationship Diagram, 实体关系图) 层、DIS (Data Item Set, 数据项集) 层和物理层 (Physical Model, 物理模型)。

ERD 层是数据模型的最高层,该层描述了公司业务中的实体或主题域以及它们之间的关系;ERD 层是中间层,该层描述了数据模型中的关键字、属性以及细节数据之间的关系;物理层是数据建模的最底层,该层描述了数据模型的物理特性。

Inmon 对于构建数据仓库模型建议采用螺旋式开发方法,采用迭代方式完成多次需求。但需要采用统一的 ERD 模型,才能够将每次迭代的结果整合在一起。ERD 模型是高度抽象的数据模型,描述了企业完整的数据。而每次迭代则是完成 ERD 模型的子集,通过 DIS 和物理数据模型实现。

上述内容主要引用自 Inmon 的 *Building the Data Warehouse*,具体细节请参考原著作。

### 3. 其他模型实施过程

在实践中经常会用到如下数据仓库模型层次的划分,和 Kimball、Inmon 的模型实施理论有一定的相通性,但不涉及具体的模型表达。

- 业务建模,生成业务模型,主要解决业务层面的分解和程序化。

- 领域建模，生成领域模型，主要是对业务模型进行抽象处理，生成领域概念模型。
- 逻辑建模，生成逻辑模型，主要是将领域模型的概念实体以及实体之间的关系进行数据库层次的逻辑化。
- 物理建模，生成物理模型，主要解决逻辑模型针对不同关系数据库的物理化以及性能等一些具体的技术问题。

## 9.4.2 OneData 实施过程

本节重点讲解怎么使用 OneData 这套体系和相配套的工具实施大数据系统的模型建设，在讲解中会以阿里巴巴的具体业务进行说明。

### 1. 指导方针

首先，在建设大数据数据仓库时，要进行充分的业务调研和需求分析。这是数据仓库建设的基石，业务调研和需求分析做得是否充分直接决定了数据仓库建设是否成功。其次，进行数据总体架构设计，主要是根据数据域对数据进行划分；按照维度建模理论，构建总线矩阵、抽象出业务过程和维度。再次，对报表需求进行抽象整理出相关指标体系，使用 OneData 工具完成指标规范定义和模型设计。最后，就是代码研发和运维。本文将会重点讲解物理模型设计之前（含）步骤的内容。

### 2. 实施工作流

实施工作流如图 9.11 所示。

#### (1) 数据调研

##### • 业务调研

整个阿里集团涉及的业务涵盖电商、数字娱乐、导航（高德）、移动互联网服务等领域。各个领域又涵盖多个业务线，如电商领域就涵盖了 C 类（淘宝、天猫、天猫国际）与 B 类（阿里巴巴中文站、国际站、速卖通）业务。数据仓库是要涵盖所有业务领域，还是各个业务领域独自建设，业务领域内的业务线也同样面临着这个问题。所以要构建大数据数据仓库，就需要了解各个业务领域、业务线的业务有什么共同点和

不同点,以及各个业务线可以细分为哪几个业务模块,每个业务模块具体的业务流程又是怎样的。业务调研是否充分,将会直接决定数据仓库建设是否成功。

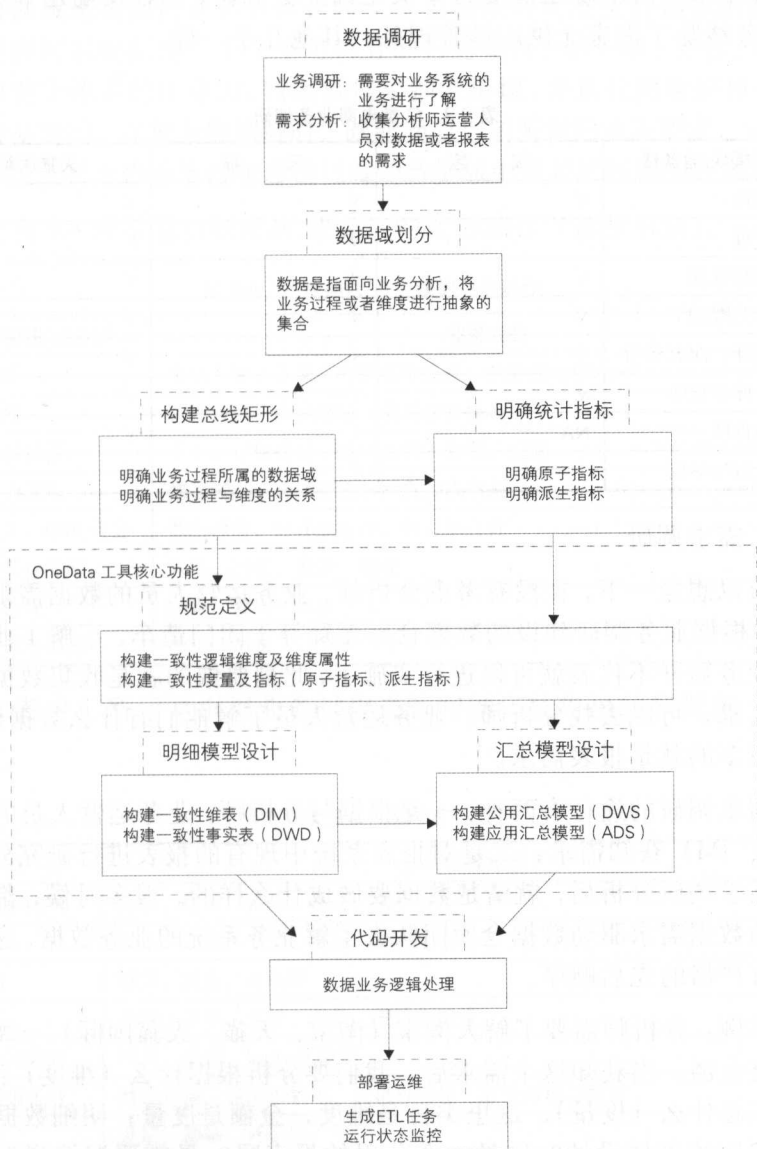


图 9.11 实施工作流



在阿里巴巴，一般各个业务领域独自建设数据仓库，业务领域内的业务线由于业务相似、业务相关性较大，进行统一集中建设。

如表 9.3 所示是粗粒度的 C 类电商业务调研，不难发现几个功能模块/业务线除了淘宝无供应链管理外，其他几乎一样。

表 9.3 C 类电商业务调研

功能模块/业务线	淘 宝	天 猫	天猫国际
商品管理	√	√	√
会员管理	√	√	√
交易流程管理	√	√	√
用户行为跟踪	√	√	√
好中差评、DSR 评分	√	√	√
物流、库存管理	√	√	√
供应链管理	NA	√	√
客户投诉与举报	√	√	√

• 需求调研

可以想象一下，在没有考虑分析师、业务运营人员的数据需求的情况下，根据业务调研建设的数据仓库无疑等于闭门造车。了解了业务系统的业务后并不代表就可以进行实施了，此刻要做的就是收集数据使用者的需求，可以去找分析师、业务运营人员了解他们有什么数据诉求，此时更多的就是报表需求。

需求调研的途径有两种：一是根据与分析师、业务运营人员的沟通（邮件、IM）获知需求；二是对报表系统中现有的报表进行研究分析。通过需求调研分析后，就清楚数据要做成什么样的。很多时候，都是由具体的数据需求驱动数据仓库团队去了解业务系统的业务数据，这两者并没有严格的先后顺序。

举例：分析师需要了解大淘宝（淘宝、天猫、天猫国际）一级类目的成交金额。当获知这个需求后，我们要分析根据什么（维度）汇总，以及汇总什么（度量），这里类目是维度，金额是度量；明细数据和汇总数据应该怎样设计？这是一个公用的报表吗？是需要沉淀到汇总表里面，还是在报表工具中进行汇总？

## (2) 架构设计

### • 数据域划分

数据域是指面向业务分析,将业务过程或者维度进行抽象的集合。业务过程可以概括为一个个不可拆分的行为事件,如下单、支付、退款。为保障整个体系的生命力,数据域需要抽象提炼,并且长期维护和更新,但不轻易变动。在划分数据域时,既能涵盖当前所有的业务需求,又能在新业务进入时无影响地被包含进已有的数据域中或者扩展新的数据域。

如表 9.4 所示是功能模块/业务线的业务动作(部分示例)。

表 9.4 功能模块/业务线的业务动作

功能模块/业务线	业务动作
商品管理	商品上架、下架、商品名称修改、商品类目修改
会员管理	新增会员、会员登录、会员信息修改
交易流程管理	下单、订单支付、确认收货、退货、退款
用户行为跟踪	商品浏览、店铺浏览、网页区块点击
好评中差评、DSR 评分	做出评价,好评改差评,给订单打分
物流、库存管理	入库、出库、发货、签收
供应链管理	采购、发货、入库
客户投诉与举报	投诉、举报

如表 9.5 所示是根据业务过程进行归纳,抽象出的数据域(部分示例)。

表 9.5 数据域

数 据 域	业务过程举例
会员和店铺域	注册、登录、装修、开店、关店等
商品域	发布、上架、下架、重发、SKU 存量等
日志域	曝光、浏览、点击等
交易域	加购、下单、支付、退款、确认收货等
客服和销售域	拜访、培训、leads 管理等
工具和服务域	商品收藏、淘金币领用、优惠券领用、服务市场订购等
互动域	发帖、回帖、评论等
信用风控域	评价、申诉、投诉、纠纷、买家保障、认证等
采购分销域	商品采购(供应链管理)

### • 构建总线矩阵

在进行充分的业务调研和需求调研后，就要构建总线矩阵了。需要做两件事情：明确每个数据域下有哪些业务过程；业务过程与哪些维度相关，并定义每个数据域下的业务过程和维度。

如表 9.6 所示是供应链管理业务过程示例。

表 9.6 供应链管理业务过程示例

数据域 业务过程		一致性维度							
		供应商	业务类型	地区	仓库	类目	采购单	发货单	入库单
采购 分销域	采购	√	√	√	√	√	√		
	发货	√	√	√	√	√		√	
	入库	√	√	√	√	√			√

### (3) 规范定义

规范定义主要定义指标体系，包括原子指标、修饰词、时间周期和派生指标。

### (4) 模型设计

模型设计主要包括维度及属性的规范定义，维表、明细事实表和汇总事实表的模型设计。相关实践详解请参考后续章节。

### (5) 总结

OneData 的实施过程是一个高度迭代和动态的过程，一般采用螺旋式实施方法。在总体架构设计完成之后，开始根据数据域进行迭代式模型设计和评审。在架构设计、规范定义和模型设计等模型实施过程中，都会引入评审机制，以确保模型实施过程的正确性。

# 第10章

## 维度设计

### 10.1 维度设计基础

#### 10.1.1 维度的基本概念

维度是维度建模的基础和灵魂。在维度建模中,将度量称为“事实”,将环境描述为“维度”,维度是用于分析事实所需要的多样环境。例如,在分析交易过程时,可以通过买家、卖家、商品和时间等维度描述交易发生的环境。

维度所包含的表示维度的列,称为维度属性。维度属性是查询约束条件、分组和报表标签生成的基本来源,是数据易用性的关键。例如,在查询请求中,获取某类目的商品、正常状态的商品等,是通过约束商品类目属性和商品状态属性来实现的;统计淘宝不同商品类目的每日成交金额,是通过商品维度的类目属性进行分组的;我们在报表中看到的类目、BC 类型(B 指天猫,C 指集市)等,都是维度属性。所以维度的作用一般是查询约束、分类汇总以及排序等。

如何获取维度或维度属性？如上面所提到的，一方面，可以在报表中获取；另一方面，可以在和业务人员的交谈中发现维度或维度属性。因为它们经常出现在查询或报表请求中的“按照”（by）语句内。例如，用户要“按照”月份和产品来查看销售情况，那么用来描述其业务的自然方法应该作为维度或维度属性包括在维度模型中。

维度使用主键标识其唯一性，主键也是确保与之相连的任何事实表之间存在引用完整性的基础。主键有两种：代理键和自然键，它们都是用于标识某维度的具体值。但代理键是不具有业务含义的键，一般用于处理缓慢变化维；自然键是具有业务含义的键。比如商品，在 ETL 过程中，对于商品维表的每一行，可以生成一个唯一的代理键与之对应；商品本身的自然键可能是商品 ID 等。其实对于前台应用系统来说，商品 ID 是代理键；而对于数据仓库系统来说，商品 ID 则属于自然键。

## 10.1.2 维度的基本设计方法

维度的设计过程就是确定维度属性的过程，如何生成维度属性，以及所生成的维度属性的优劣，决定了维度使用的方便性，成为数据仓库易用性的关键。正如 Kimball 所说的，数据仓库的能力直接与维度属性的质量和深度成正比。

下面以淘宝的商品维度为例对维度设计方法进行详细说明。

第一步：选择维度或新建维度。作为维度建模的核心，在企业级数据仓库中必须保证维度的唯一性。以淘宝商品维度为例，有且只允许有一个维度定义。

第二步：确定主维表。此处的主维表一般是 ODS 表，直接与业务系统同步。以淘宝商品维度为例，s\_auction\_auctions 是与前台商品中心系统同步的商品表，此表即是主维表。

第三步：确定相关维表。数据仓库是业务源系统的数据整合，不同业务系统或者同一业务系统中的表之间存在关联性。根据对业务的梳理，确定哪些表和主维表存在关联关系，并选择其中的某些表用于生成维度属性。以淘宝商品维度为例，根据对业务逻辑的梳理，可以得到商

品与类目、SPU、卖家、店铺等维度存在关联关系。

第四步：确定维度属性。本步骤主要包括两个阶段，其中第一个阶段是从主维表中选择维度属性或生成新的维度属性；第二个阶段是从相关维表中选择维度属性或生成新的维度属性。以淘宝商品维度为例，从主维表（s\_auction\_auctions）和类目、SPU、卖家、店铺等相关维表中选择维度属性或生成新的维度属性。

确定维度属性的几点提示：

#### （1）尽可能生成丰富的维度属性

比如淘宝商品维度有近百个维度属性，为下游的数据统计、分析、探查提供了良好的基础。

#### （2）尽可能多地给出包括一些富有意义的文字性描述

属性不应该是编码，而应该是真正的文字。在阿里巴巴维度建模中，一般是编码和文字同时存在，比如商品维度中的商品 ID 和商品标题、类目 ID 和类目名称等。ID 一般用于不同表之间的关联，而名称一般用于报表标签。

#### （3）区分数值型属性和事实

数值型字段是作为事实还是维度属性，可以参考字段的一般用途。如果通常用于查询约束条件或分组统计，则是作为维度属性；如果通常用于参与度量的计算，则是作为事实。比如商品价格，可以用于查询约束条件或统计价格区间的商品数量，此时是作为维度属性使用的；也可以用于统计某类目下商品的平均价格，此时是作为事实使用的。另外，如果数值型字段是离散值，则作为维度属性存在的可能性较大；如果数值型字段是连续值，则作为度量存在的可能性较大，但并不绝对，需要同时参考字段的具体用途。

#### （4）尽量沉淀出通用的维度属性

有些维度属性获取需要进行比较复杂的逻辑处理，有些需要通过多表关联得到，或者通过单表的不同字段混合处理得到，或者通过对单表的某个字段进行解析得到。此时，需要将尽可能多的通用的维度属性进行沉淀。一方面，可以提高下游使用的方便性，减少复杂度；另一方面，

可以避免下游使用解析时由于各自逻辑不同而导致口径不一致。例如，淘宝商品的 `property` 字段，使用 `key:value` 方式存储多个商品属性。商品品牌就存存储在此字段中，而商品品牌是重要的分组统计和查询约束的条件，所以需要将品牌解析出来，作为品牌属性存在。例如，商品是否在线，即在淘宝网站是否可以查看到此商品，是重要的查询约束的条件，但是无法直接获取，需要进行加工，加工逻辑是：商品状态为 0 和 1 且商品上架时间小于或等于当前时间，则是在线商品；否则是非在线商品。所以需要封装商品是否在线的逻辑作为一个单独的属性字段。

### 10.1.3 维度的层次结构

维度中的一些描述属性以层次方式或一对多的方式相互关联，可以被理解为包含连续主从关系的属性层次。层次的最底层代表维度中描述最低级别的详细信息，最高层代表最高级别的概要信息。维度常常有多个这样的嵌入式层次结构。比如淘宝商品维度，有卖家、类目、品牌等。商品属于类目，类目属于行业，其中类目的最低级别是叶子类目，叶子类目属于二级类目，二级类目属于一级类目。

在属性的层次结构中进行钻取是数据钻取的方法之一。关于钻取的定义，这里不做介绍，读者可以参考相关书籍。下面通过具体的例子，看看如何在层次结构中进行钻取。

假设已有一个淘宝交易订单，创建事实表。现在统计 2015 年“双 11”的下单 GMV，得到一行记录；沿着层次向下钻取，添加行业，得到行业实例个数的记录数；继续沿着层次向下钻取，添加一级类目，得到一级类目实例个数的记录数。可以看到，通过向报表中添加连续的维度细节级别，实现在层次结构中进行钻取。

- 最高层次的统计，如表 10.1 所示。

表 10.1 最高层次统计

日 期	行 业	一级类目	下单 GMV
20151111	ALL	ALL	912 亿

- 钻取至行业层次，统计如表 10.2 所示。

表 10.2 钻取至行业层次统计

日 期	行 业	一级类目	下单 GMV
20151111	行业 1	ALL	industry1_gmv
	行业 2	ALL	industry2_gmv
	...	...	...
	行业 N	ALL	industryN_gmv

- 钻取至一级类目层次，统计如表 10.3 所示。

表 10.3 钻取至一级类目层次统计

日 期	行 业	一级类目	下单 GMV
20151111	行业 1	cat1	industry1_cat1_gmv
	...	...	...
	行业 1	catM	industry1_catM_gmv
	行业 2	catX	industry2_catX_gmv
	...	...	...
	行业 2	catY	industry2_catY_gmv
	...	...	...

类目、行业、品牌等属性层次是被实例化为多个维度，还是作为维度属性存在于商品维度中？如何设计，我们在下一节中详细讨论。

#### 10.1.4 规范化和反规范化

当属性层次被实例化为一系列维度，而不是单一的维度时，被称为雪花模式。大多数联机事务处理系统（OLTP）的底层数据结构在设计时采用此种规范化技术，通过规范化处理将重复属性移至其自身所属的表中，删除冗余数据。

这种方法用在 OLTP 系统中可以有效避免数据冗余导致的不一致性。比如在 OLTP 系统中，存在商品表和类目表，且商品表中有冗余的



类目表的属性字段，假设对某类目进行更新，则必须更新商品表和类目表，且由于商品和类目是一对多的关系，商品表可能每次需要更新几十万甚至上百万条记录，这是不合理的。而对于联机分析处理系统(OLAP)来说，数据是稳定的，不存在 OLTP 系统中所存在的问题。

对于淘系商品维度，如果采用雪花模式进行规范化处理，将表现为如图 10.1 所示的形式。

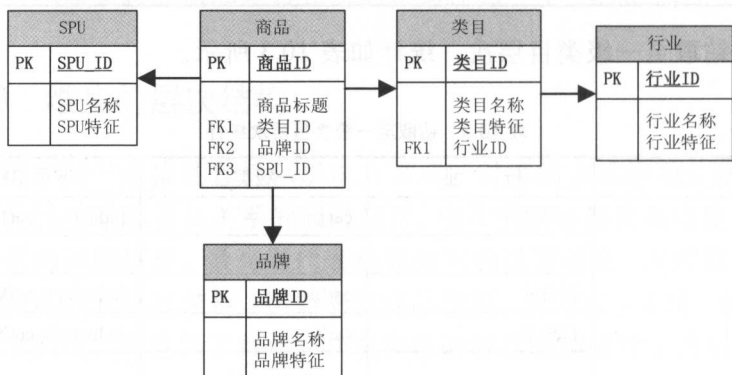


图 10.1 规范化处理淘宝商品维度所表现的形式

将维度的属性层次合并到单个维度中的操作称为反规范化。分析系统的主要目的是用于数据分析和统计，如何更方便用户进行统计分析决定了分析系统的优劣。采用雪花模式，用户在统计分析的过程中需要大量的关联操作，使用复杂度高，同时查询性能很差；而采用反规范化处理，则方便、易用且性能好。

对于淘宝商品维度，如果采用反规范化处理，将表现为如图 10.2 所示的形式。

如上所述，从用户角度来看简化了模型，并且使数据库查询优化器的连接路径比完全规范化的模型简化许多。反规范化的维度仍包含与规范化模型同样的信息和关系，从分析角度来看，没有丢失任何信息，但复杂性降低了。

商品	
PK	商品ID
	商品标题 类目名称 类目特征 行业名称 行业特征 品牌名称 品牌特征 SPU名称 SPU特征

图 10.2 反规范化处理淘宝商品维度所表现的形式

采用雪花模式，除了可以节约一部分存储外，对于 OLAP 系统来说没有其他效用。而现阶段存储的成本非常低。出于易用性和性能的考虑，维表一般是很不规范化的。在实际应用中，几乎总是使用维表的空间来换取简明性和查询性能。

### 10.1.5 一致性维度和交叉探查

构建企业级数据仓库不可能一蹴而就，一般采用迭代式的构建过程。而单独构建存在的问题是形成独立型数据集市，导致严重的不一致性。Kimball 的数据仓库总线架构提供了一种分解企业级数据仓库规划任务的合理方法，通过构建企业范围内一致性维度和事实来构建总线架构。

数据仓库总线架构的重要基石之一就是一致性维度。在针对不同数据域进行迭代构建或并行构建时，存在很多需求是对于不同数据域的业务过程或者同一数据域的不同业务过程合并在一起观察。比如对于日志数据域，统计了商品维度的最近一天的 PV 和 UV；对于交易数据域，统计了商品维度的最近一天的下单 GMV。现在将不同数据域的商品的事实合并在一起进行数据探查，如计算转化率等，称为交叉探查。

如果不同数据域的计算过程使用的维度不一致，就会导致交叉探查存在问题。当存在重复的维度，但维度属性或维度属性的值不一致时，会导致交叉探查无法进行或交叉探查结果错误。接上个例子，假设对于日志数据域，统计使用的是商品维度 1；对于交易数据域，统计使用的是商品维度 2。商品维度 1 包含维度属性 BC 类型，而商品维度 2 无此

属性,则无法在 BC 类型上进行交叉探查;商品维度 1 的商品上架时间这一维度属性时间格式是 yyyy-MM-dd HH:mm:ss,商品维度 2 的商品上架时间这一维度属性时间格式是 UNIX timestamp,进行交叉探查时如果需要根据商品上架时间做限制,则复杂性较高;商品维度 1 不包含阿里旅行的商品,商品维度 2 包含全部的淘系商品,交叉探查也无法进行。还有很多种形式的不一致,这里不再一一列举,但基本可以划分为维度格式和内容不一致两种类型。

上面对维度不一致性进行了详细分析,下面总结维度一致性的几种表现形式。

- 共享维表。比如在阿里巴巴的数据仓库中,商品、卖家、买家、类目等维度有且只有一个。所以基于这些公共维度进行的交叉探查不会存在任何问题。
- 一致性上卷,其中一个维度的维度属性是另一个维度的维度属性的子集,且两个维度的公共维度属性结构和内容相同。比如在阿里巴巴的商品体系中,有商品维度和类目维度,其中类目维度的维度属性是商品维度的维度属性的子集,且有相同的维度属性和维度属性值。这样基于类目维度进行不同业务过程的交叉探查也不会存在任何问题。
- 交叉属性,两个维度具有部分相同的维度属性。比如在商品维度中具有类目属性,在卖家维度中具有主营类目属性,两个维度具有相同的类目属性,则可以在相同的类目属性上进行不同业务过程的交叉探查。

## 10.2 维度设计高级主题

### 10.2.1 维度整合

我们先来看数据仓库的定义:数据仓库是一个面向主题的、集成的、非易失的且随时间变化的数据集合,用来支持管理人员的决策。其中集

成是数据仓库的四个特性中最重要的一个。

数据仓库的重要数据来源是大量的、分散的面向应用的操作型环境。不同的应用在设计过程中,可以自由决策,主要满足本应用的需求,很少会考虑和其他系统进行数据集成。应用之间的差异具体表现在如下几个方面:

- 应用在编码、命名习惯、度量单位等方面会存在很大的差异。比如不同应用对于用户的性别编码不同,有0和1、F和M等;不同应用的用户ID含义相同,但字段名称不同,有user、user\_id等;不同应用对于金额的度量单位不同,有元、分等。
- 应用出于性能和扩展性的考虑,或者随技术架构的演变,以及业务的发展,采用不同的物理实现。拆分至不同类型数据库中,部分数据采用关系型数据库存储(如Oracle、MySQL等),部分数据采用NoSQL数据库存储(如HBase、Tair等)。拆分成同一类型数据库中的多个物理表,比如对于淘宝商品,有商品主表和商品扩展表,商品主表存储商品基本信息,商品扩展表存储商品特殊信息,如不同产品线的定制化信息等;对于淘宝会员,有会员主表和会员扩展表,会员主表存储用户基本信息,会员扩展表存储用户扩展信息,如用户的各种标签信息等。

所以数据由面向应用的操作型环境进入数据仓库后,需要进行数据集成。将面向应用的数据转换为面向主题的数据仓库数据,本身就是一种集成。具体体现在如下几个方面:

- 命名规范的统一。表名、字段名等统一。
- 字段类型的统一。相同和相似字段的字段类型统一。
- 公共代码及代码值的统一。公共代码及标志性字段的数据类型、命名方式等统一。
- 业务含义相同的表的统一。主要依据高内聚、低耦合的理念,在物理实现中,将业务关系大、源系统影响差异小的表进行整合;将业务关系小、源系统影响差异大的表进行分而置之。通常有如下几种集成方式:

➤ 采用主从表的设计方式,将两个表或多个表都有的字段放在

主表中（主要基本信息），从属信息分别放在各自的从表中。对于主表中的主键，要么采用复合主键、源主键和系统或表区别标志；要么采用唯一主键、“源主键和系统或表区别标志”生成新的主键。通常建议采用复合主键的方式。

- 直接合并，共有信息和个性信息都放在同一个表中。如果表字段的重合度较低，则会出现大量空值，对于存储和易用性会有影响，需谨慎选择。
- 不合并，因为源表的表结构及主键等差异很大，无法合并，使用数据仓库里的多个表存放各自的数据。

维表的整合涉及的内容和上面介绍的几个方面相同，下面重点看表级别的整合，有两种表现形式。

第一种是垂直整合，即不同的来源表包含相同的数据集，只是存储的信息不同。比如淘宝会员在源系统中有多多个表，如会员基础信息表、会员扩展信息表、淘宝会员等级信息表、天猫会员等级信息表，这些表都属于会员相关信息表，依据维度设计方法，尽量整合至会员维度模型中，丰富其维度属性。

第二种是水平整合，即不同的来源表包含不同的数据集，不同子集之间无交叉，也可以存在部分交叉。比如针对蚂蚁金服的数据仓库，其采集的会员数据有淘宝会员、1688 会员、国际站会员、支付宝会员等，是否需要将所有的会员整合到一个会员表中呢？如果进行整合，首先需要考虑各个会员体系是否有交叉，如果存在交叉，则需要去重；如果不存在交叉，则需要考虑不同子集的自然键是否存在冲突，如果不冲突，则可以考虑将各子集的自然键作为整合后的表的自然键；另一种方式是设置超自然键，将来源表各子集的自然键加工成一个字段作为超自然键。在阿里巴巴，通常采用将来源表各子集的自然键作为联合主键的方式，并且在物理实现时将来源字段作为分区字段。

有整合就有拆分，到底是整合还是拆分，由多种因素决定。下面两节讨论维度的水平拆分和垂直拆分。

## 10.2.2 水平拆分

维度通常可以按照类别或类型进行细分。比如淘系商品表,根据业务线或行业等可以对商品进行细分,如淘宝的商品、天猫的商品、1688的商品、飞猪旅行的商品、淘宝海外的商品、天猫国际的商品等。不同分类的商品,其维度属性可能相同,也可能不同。比如航旅的商品和普通的淘系商品,都属于商品,都有商品价格、标题、类型、上架时间、类目等维度属性,但是航旅的商品除了有这些公共属性外,还有酒店、景点、门票、旅行等自己独特的维度属性。

如何设计维度?针对此问题,主要有两种解决方案:方案1是将维度的不同分类实例化为不同的维度,同时在主维度中保存公共属性;方案2是维护单一维度,包含所有可能的属性。

选择哪种方案?在数据模型设计过程中需要考虑的因素有很多,基本不可能满足各个特性指标的最优化。在设计过程中需要重点考虑以下三个原则。

- 扩展性:当源系统、业务逻辑变化时,能通过较少的成本快速扩展模型,保持核心模型的相对稳定性。软件工程中的高内聚、低耦合的思想是重要的指导方针之一。
- 效能:在性能和成本方面取得平衡。通过牺牲一定的存储成本,达到性能和逻辑的优化。
- 易用性:模型可理解性高、访问复杂度低。用户能够方便地从模型中找到对应的数据表,并能够方便地查询和分析。

根据数据模型设计思想,在对维度进行水平拆分时,主要考虑如下两个依据。

第一个依据是维度的不同分类的属性差异情况。当维度属性随类型变化较大时,将所有可能的属性建在一个表中是不切合实际的,也没有必要这样做,此时建议采用方案1。定义一个主维度用于存放公共属性;同时定义多个子维度,其中除了包含公共属性外,还包含各自的特殊属性。比如在阿里巴巴数据仓库维度体系中,依据此方法,构建了商品维度、航旅商品维度等。公共属性一般比较稳定,通过核心的商品维

度,保证了核心维度的稳定性;通过扩展子维度的方式,保证了模型的扩展性。

第二个依据是业务的关联程度。两个相关性较低的业务,耦合在一起弊大于利,对模型的稳定性和易用性影响较大。比如在阿里巴巴数据仓库维度体系中,对淘系商品和 1688 商品构建两个维度。虽然淘系和 1688 在底层技术实现上是统一的,但属于不同的 BU,业务各自发展;在数据仓库层面,淘系和 1688 属于不同的数据集市,一般不会相互调用,业务分析人员一般只针对本数据集市进行统计分析。如果设计成一个维度,由于不同 BU 业务各自发展,1688 业务变更,此维度需要变更,淘宝业务变更亦然,稳定性很差;在易用性方面,会给数据使用方造成困扰。

### 10.2.3 垂直拆分

在维度设计内容中,我们提到维度是维度建模的基础和灵魂,维度属性的丰富程度直接决定了数据仓库的能力。在进行维度设计时,依据维度设计的原则,尽可能丰富维度属性,同时进行反规范化处理。对于具体实现时可能存在的问题,一是在“水平拆分”中提到的,由于维度分类的不同而存在特殊的维度属性,可以通过水平拆分的方式解决此问题。

二是某些维度属性的来源表产出时间较早,而某些维度属性的来源表产出时间较晚;或者某些维度属性的热度高、使用频繁,而某些维度属性的热度低、较少使用;或者某些维度属性经常变化,而某些维度属性比较稳定。在“水平拆分”中提到的模型设计的三个原则同样适合解决此问题。

出于扩展性、产出时间、易用性等方面的考虑,设计主从维度。主维表存放稳定、产出时间早、热度高的属性;从维表存放变化较快、产出时间晚、热度低的属性。比如在阿里巴巴数据仓库中,设计了商品主维度和商品扩展维度。其中商品主维度在每日的 1:30 左右产出,而商品扩展维度由于有冗余的产出时间较晚的商品品牌和标签信息,在每日的 3:00 左右产出。另外,由于商品扩展维度有冗余的库存等变化较快的数据,对于主维度进行缓慢变化的处理较为重要。通过存储的冗余和



计算成本的增加,实现了商品主模型的稳定和产出时间的提前,对于整个数据仓库的稳定和下游应用的产出都有较大意义。

#### 10.2.4 历史归档

阿里巴巴历史截至当前的淘系(含淘宝、天猫和聚划算)商品有几百亿条记录,在 MaxCompute 中,一天的全量数据占用约 36TB 的存储。面对如此庞大的数据量,如何设计模型、如何降低存储、如何让下游方便获取数据,成为必须要解决的问题。对于历史数据,是否存在前台已经不再使用的情况?答案是肯定的,对于如此庞大的数据量,现有的技术架构也很难处理。前台有一套数据归档策略,比如将商品状态为下架或删除的且最近 31 天未更新的商品归档至历史库;具体逻辑根据不同 BU 有不同的算法,且有特殊的规则。

在数据仓库中,可以借用前台数据库的归档策略,定期将历史数据归档至历史维表。在实践中,阿里巴巴数据仓库设计了商品维表和历史商品维表,每天将历史数据归档至历史商品维表。关于归档策略,有以下几种方式。

归档策略 1: 同前台归档策略,在数据仓库中实现前台归档算法,定期对历史数据进行归档。但存在一些问题,一是前台归档策略复杂,实现成本较高;二是前台归档策略可能会经常变化,导致数据仓库归档算法也要随之变化,维护和沟通成本较高。此方式适用于前台归档策略逻辑较为简单,且变更不频繁的情况。

归档策略 2: 同前台归档策略,但采用数据库变更日志的方式。对于如此庞大的数据量,阿里巴巴采用的数据抽取策略一般是通过数据库 binlog 日志解析获取每日增量,通过增量 merge 全量的方式获取最新的全量数据。可以使用增量日志的删除标志,作为前台数据归档的标志。通过此标志对数据仓库的数据进行归档。此方式不需要关注前台归档策略,简单易行。但对前台应用的要求是数据库的物理删除只有在归档时才执行,应用中的删除只是逻辑删除。

归档策略 3: 数据仓库自定义归档策略。可以将归档算法用简单、



直接的方式实现，但原则是尽量比前台应用晚归档、少归档。避免出现数据仓库中已经归档的数据再次更新的情况。

如果技术条件允许，能够解析数据库 binlog 日志，建议使用归档策略 2，规避前台归档算法。具体可以根据自身数据仓库的实际情况进行选择。

## 10.3 维度变化

### 10.3.1 缓慢变化维

数据仓库的重要特点之一是反映历史变化，所以如何处理维度的变化是维度设计的重要工作之一。缓慢变化维的提出是因为在现实世界中，维度的属性并不是静态的，它会随着时间的流逝发生缓慢的变化。与数据增长较为快速的事实表相比，维度变化相对缓慢。

在一些情况下，保留历史数据没有什么分析价值；而在另一些情况下，保留历史数据将会起到至关重要的作用。在 Kimball 的理论中，有三种处理缓慢变化维的方式，下面通过简单的实例进行说明，具体细节请翻阅 Kimball 的相关书籍。

第一种处理方式：重写维度值。采用此种方式，不保留历史数据，始终取最新数据。比如，商品所属的类目于 2015 年 11 月 16 日由类目 1 变成类目 2，采用第一种处理方式，变化前后的数据记录分别如表 10.4 和表 10.5 所示。

表 10.4 变化前商品表和订单表

商品 Key	商品 ID	商品标题	所属类目	其他维度属性
1000	item1	title1	类目 1	...

订单 Key	日期 Key	商品 Key	交易金额	其他事实
9000	2015-11-11	1000	103.00	...

表 10.5 变化后商品表和订单表

商品 Key	商品 ID	商品标题	所属类目	其他维度属性
1000	item1	titile1	类目 2	...

订单 Key	日期 Key	商品 Key	交易金额	其他事实
9000	2015-11-11	1000	103.00	...
9001	2015-11-16	1000	89.00	...

第二种处理方式：插入新的维度行。采用此种方式，保留历史数据，维度值变化前的事实和过去的维度值关联，维度值变化后的事实和当前的维度值关联。同上面的例子，采用第二种处理方式，变化前的数据记录同表 10.4，变化后的数据记录如表 10.6 所示。

表 10.6 变化后商品表和订单表

商品 Key	商品 ID	商品标题	所属类目	其他维度属性
1000	Item1	titile1	类目 1	...
1001	Item1	titile1	类目 2	...

订单 Key	日期 Key	商品 Key	交易金额	其他事实
9000	2015-11-11	1000	103.00	...
9001	2015-11-16	1001	89.00	...

第三种处理方式：添加维度列。采用第二种处理方式不能将变化前后记录的事实归一为变化前的维度或者归一为变化后的维度。比如根据业务需求，需要将 11 月份的交易金额全部统计到类目 2 上，采用第二种处理方式无法实现。针对此问题，采用第三种处理方式，保留历史数据，可以使用任何一个属性列。同上面的例子，采用第三种处理方式，变化前后的数据记录分别如表 10.7 和表 10.8 所示。通过变化后的商品表和订单表关联，可以根据不同的业务需求，将 11 月份的交易金额全部统计到类目 2 或类目 1 上。

表 10.7 变化前商品表和订单表

商品 Key	商品 ID	商品标题	所属新类目	所属旧类目	其他维度属性
1000	item1	titile1	类目 1	类目 1	...

订单 Key	日期 Key	商品 Key	交易金额	其他事实
9000	2015-11-11	1000	103.00	...

表 10.8 变化后商品表和订单表

商品 Key	商品 ID	商品标题	所属新类目	所属旧类目	其他维度属性
1000	item1	titile1	类目 2	类目 1	...

订单 Key	日期 Key	商品 Key	交易金额	其他事实
9000	2015-11-11	1000	103.00	...
9001	2015-11-16	1000	89.00	...

对于选择哪种方式处理缓慢变化维，并没有一个完全正确的答案，可以根据业务需求来进行选择。比如根据商品所属的类目统计淘宝 2015 年 11 月的成交额，商品所属的类目于 2015 年 11 月 16 日由类目 1 变成类目 2，假设业务需求方不关心历史数据，将所有的成交额都统计到最新的类目 2 上，则不需要保存历史数据；假设类目 1 属于某个业务部门，类目 2 属于另一个业务部门，不同业务部门需要统计各自的业绩，则需要保留历史数据。

### 10.3.2 快照维表

在“维度的基本概念”中，介绍了自然键和代理键的定义，在 Kimball 的维度建模中，必须使用代理键作为每个维表的主键，用于处理缓慢变化维。

但在阿里巴巴数据仓库建设的实践过程中，虽然使用的是 Kimball 的维度建模理论，但实际并未使用代理键。那么为什么不使用代理键？如何处理缓慢变化维？

首先看“为什么不使用代理键”这个问题。第一个原因是，阿里巴巴数据量庞大，使用的是阿里巴巴自助知识产权的分布式计算平台 MaxCompute。对于分布式计算系统，不存在事务的概念，对于每个表的记录生成稳定的全局唯一的代理键难度很大，此处稳定指某条记录每

次生成的代理键都相同。第二个原因是，使用代理键会大大增加 ETL 的复杂性，对 ETL 任务的开发和维护成本很高。

接下来讨论不使用代理键如何处理缓慢变化维的问题。在阿里巴巴数据仓库实践中，处理缓慢变化维的方法是采用快照方式。数据仓库的计算周期一般是每天一次，基于此周期，处理维度变化的方式就是每天保留一份全量快照数据。比如商品维度，每天保留一份全量商品快照数据。任意一天的事实均可以获取到当天的商品信息，也可以获取到最新的商品信息，通过限定日期，采用自然键进行关联即可。此方法既有优点，也有弊端。

优点主要有以下两点：

- 简单而有效，开发和维护成本低。
- 使用方便，理解性好。数据使用方只需要限定日期，即可获取到当天的快照数据。任意一天的事实快照和维度快照通过维度的自然键进行关联即可。

弊端主要体现在存储的极大浪费上。比如某维度，每天的变化量占总体数据量的比例很低，在极端情况下，每天无变化，使得存储浪费很严重。此方法主要就是实现了牺牲存储获取 ETL 效率的优化和逻辑上的简化。但是一定要杜绝过度使用这种方法，而且必须要有对应的数据生命周期制度，清除无用的历史数据。

综合来看，由于现在存储成本远低于 CPU、内存等的成本，此方法弊大于利。那么是否有方法既可以实现上面的优点，同时又很好地降低存储呢？答案是肯定的，那就是阿里巴巴的极限存储。

### 10.3.3 极限存储

首先来看历史拉链存储。历史拉链存储是指利用维度模型中缓慢变化维的第二种处理方式。这种处理方式是通过新增两个时间戳字段（start\_dt 和 end\_dt），将所有以天为粒度的变更数据都记录下来。通常分区字段也是时间戳字段。

例如，2016 年 1 月 1 日，卖家 A 在淘宝网发布了 B、C 两个商品，前端商品表将生成两条记录 t1、t2；1 月 2 日，卖家 A 将 B 商品下架了，同时又发布了商品 D，前端商品表将更新记录 t1，又新生成记录 t3；采用全量存储方式，在 1 月 1 日这个分区中存储 t1 和 t2 两条记录；在 1 月 2 日这个分区中存储更新后的 t1 以及 t2、t3 记录。数据存储记录如表 10.9 所示。

表 10.9 数据存储记录 1

商品	dt	卖家	状态	其他字段
B	20160101	A	上架	...
C	20160101	A	上架	...
B	20160102	A	下架	...
C	20160102	A	上架	...
D	20160102	A	上架	...

如果采用历史拉链存储，数据存储记录如表 10.10 所示。对于不变的数据，不再重复存储。

表 10.10 数据存储记录 2

商品	start_dt	end_dt	卖家	状态	其他字段
B	20160101	20160102	A	上架	...
C	20160101	30001231	A	上架	...
B	20160102	30001231	A	下架	...
D	20160102	30001231	A	上架	...

这样下游应用可以通过限制时间戳字段来获取历史数据。例如，用户访问 1 月 1 日的数据，只需要限制 `start_dt<=20160101` 和 `end_dt>20160101` 即可。

但是这种存储方式对于下游使用方存在一定的理解障碍，特别是 ODS 数据面向的下游用户包括数据分析师、前端开发人员等，他们不怎么理解维度模型的概念，因此会存在较高的解释成本。另外，这种存储方式用 `start_dt` 和 `end_dt` 做分区，随着时间的推移，分区数量会极度膨胀，而现行的数据库系统都有分区数量限制。

为了解决上述两个问题，阿里巴巴提出采用极限存储的方式来处理。

## 1. 透明化

底层的数据还是历史拉链存储，但是上层做一个视图操作或者在Hive里做一个hook，通过分析语句的语法树，把对极限存储前的表的查询转换成对极限存储表的查询。对于下游用户来说，极限存储表和全量存储方式是一样的：

```
Select * from A where ds =20160101 ;
```

等价于

```
Select * from A_EXST where start_dt <=20160101 and end_dt  
>20160101;
```

## 2. 分月做历史拉链表

假设用 start\_dt 和 end\_dt 做分区，并且不做限制，那么可以计算出一年历史拉链表最多可能产生的分区数是： $365 \times 364 / 2 = 66430$  个。如果在每个月月初重新开始做历史拉链表，目录结构如下：

```
|-- 201410/                                # 每月一个周期
|---- 20141001/ 201410_INFINITY             # 每月 1 日的全量数据
|---- 20141001/20141002                     # 1001 产生且 1002 死亡记录
|---- 20141001/20141003                     # 1001 产生且 1003 死亡记录
...
|----- 20141001/201410031                  # 1001 产生且 10031 死亡记录
|----- 20141002/ 201410_INFINITY           # 1002 产生新增记录
|---- -20141002/20141003                    # 1002 产生且 1003 死亡记录
...
|---- 20141002 /201410031/                  # 1002 产生且 10031 死亡记录
|-----20141003/ 201410_INFINITY            # 1003 产生新增记录
...
|---- 20141031/ 201410_INFINITY              # 1031 产生新增记录
|-- 201411/
...
```

再计算一年最多可能产生的分区数是： $12 \times (1 + (30 + 29) / 2) = 5232$  个。

采用极限存储的处理方式，极大地压缩了全量存储的成本，又可以达到对下游用户透明的效果，是一种比较理想的存储方式。但是其本身也有一定的局限性，首先，其产出效率很低，大部分极限存储通常需要  $t-2$ ；其次，对于变化频率高的数据并不能达到节约成本的效果。因此，在实际生产中，做极限存储需要进行一些额外的处理。

- 在做极限存储前有一个全量存储表，全量存储表仅保留最近一段时间的全量分区数据，历史数据通过映射的方式关联到极限存储表。即用户只访问全量存储表，所以对用户来说极限存储是不可见的。
- 对于部分变化频率频繁的字段需要过滤。例如，用户表中存在用户积分字段，这种字段的值每天都在发生变化，如果不过滤的话，极限存储就相当于每个分区存储一份全量数据，起不到节约存储成本的效果。

#### 10.3.4 微型维度

采用极限存储，需要避免维度的过度增长。比如对于商品维表，每天 20 多亿条数据，如果在设计商品维度时，将值变化频繁的属性加入到商品维度中，极端情况是每天所有商品数据都发生变化，此时，极限存储没有意义；反之，每天所有商品数据都不发生变化，此时，只需要存储一天的数据即可。

通过将一些属性从维表中移出，放置到全新的维表中，可以解决维度的过度增长导致极限存储效果大打折扣的问题。其中一种解决方法就是上一节提到的垂直拆分，保持主维度的稳定性；另一种解决方式是采用微型维度。

微型维度的创建是通过将一部分不稳定的属性从主维度中移出，并将它们放置到拥有自己代理键的新表中来实现的。这些属性相互之间没有直接关联，不存在自然键。通过为每个组合创建新行的一次性过程来加载数据。比如淘宝用户维度，用户的注册日期、年龄、性别、身份信



息等基本不会发生变化,但用户 VIP 等级、用户信用评价等级会随着用户的行为不断发生变化。其中 VIP 等级共有 8 个值,即 1~6;用户信用评价等级共有 18 个值。假设基于 VIP 等级和用户信用评价等级构建微型维度,则在此微型维度中共有  $8 \times 18$  个组合,即 144 条记录,代理键可能是 1~144。

这里以淘宝交易事实表为例,其他维度忽略,星形模式可能表示如图 10.3 所示。

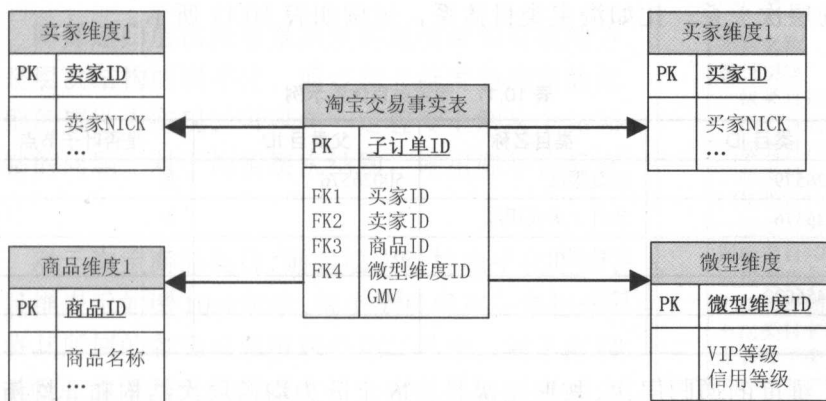


图 10.3 淘宝交易事实表维度

但在阿里巴巴数据仓库实践中,并未使用此技术,主要有以下几点原因:

- 微型维度的局限性。微型维度是事先用所有可能值的组合加载的,需要考虑每个属性的基数,且必须是枚举值。很多属性可能是非枚举型,比如数值类型,如 VIP 分数、信用分数等;时间类型,如上架时间、下架时间、变更时间等。
- ETL 逻辑复杂。对于分布式系统,生成代理键和使用代理键进行 ETL 加工都非常复杂,ETL 开发和维护成本过高。
- 破坏了维度的可浏览性。买家维度和微型维度通过事实表建立联系,无法基于 VIP 等级、信用等级进行浏览和统计。可以通过在买家维度中添加引用微型维度的外键部分来解决此问题,但带来的问题是微型维度未维护历史信息。



## 10.4 特殊维度

### 10.4.1 递归层次

上面我们学习了维度的层次结构，即维度属性以层次方式或一对多的方式相互关联；或者描述为不同维度之间的主从关系，比如商品和类目的关系、商品和品牌的关系等。本节的递归层次指的是某维度的实例值的层次关系，比如淘宝类目体系，示例如表 10.11 所示。

表 10.11 淘宝类目体系示例

类目 ID	类目名称	父类目 ID	是否叶子节点
50026579	圣诞服饰	50026576	是
50026576	节日 / 庆典用品	21	否
21	家具日用	0	否
121456022	台历	21	是
...	...	...	...

维度的递归层次，按照层级是否固定分为均衡层次结构和非均衡层次结构。比如类目，有固定数量的级别，分别是叶子类目、五级类目、四级类目、三级类目、二级类目、一级类目；地区，分别是乡镇/街道、区县、城市、省份、国家。对于这种具有固定数量级别的递归层次，称为“均衡层次结构”。比如公司之间的关系，每个公司可能存在一个母公司，但可能没有固定的一级、二级等层级关系。对于这种数量级别不固定的递归层次，称为“非均衡层次结构”。

淘宝交易事实表通过叶子类目和类目维表关联，如何统计类目 ID 为 21 的最近一天的 GMV？第一步，获取父类目 ID 等于 20 的所有类目，称为子类目。第二步，对于每个子类目，如果为叶子类目，则终止；如果非叶子类目，则此类目 ID 作为父类目 ID 执行第一步，直到找到所有叶子类目，如圣诞服饰（50026579）、台历（121456022）等。将所有叶子类目和交易事实表关联进行统计汇总，即可得到类目 ID 等于 21 的最近一天的 GMV，也就是家具日用类目的最近一天的 GMV。在物理实现时，可以使用递归 SQL 实现，如 Oracle 中的 connect by 语句。

通过数据探查得知 ID 等于 21 的类目属于一级类目（父类目 ID 等于 0），统计其最近一天的 GMV 的过程，称为上钻；在递归层次中进行上钻和下钻是很常见的。由于很多数据仓库系统和商业智能工具不支持递归 SQL，且用户使用递归 SQL 的成本较高，所以在维度模型中，需要对此层次结构进行处理。

1. 层次结构扁平化

降低递归层次使用复杂度的最简单和有效的方法是层次结构的扁平化，通过建立维度的固定数量级别的属性来实现，可以在一定程度上解决上钻和下钻的问题。对于均衡层次结构，采用扁平化最有效。

对于淘宝商品类目，通过层次结构扁平化之后，类目维表示如图 10.4 所示。每个类目保存一条记录，并将其所属的各类目层级属性化。其中，对于高层级类目，由于其无低层级类目，则低层级类目置为空值。

类目	
PK	类目ID
	类目名称
	类目层级
	一级类目ID
	一级类目名称
	二级类目ID
	二级类目名称
	三级类目ID
	三级类目名称
	四级类目ID
	四级类目名称
	五级类目ID
	五级类目名称
	是否叶子类目

图 10.4 类目维度

具体数据存储示例如表 10.12 所示，其中四级和五级类目省略。

表 10.12 数据存储实例 1

类目 ID	类目级别	是否叶子节点	一级类目	二级类目	三级类目
21	1	N	21	—	—
50026576	2	N	21	50026576	—
50026579	3	Y	21	50026576	50026579
121456022	2	Y	21	121456022	—
...	...	...	...	...	...

如何统计类目 ID 为 21 的最近一天的 GMV？将淘宝交易事实表通过叶子类目和类目维表的类目 ID 关联之后，限制一级类目 ID 等于 21 之后进行汇总统计，即可以得到类目 ID 等于 21 的最近一天的 GMV。其使用方便性得到大大提高，但存在如下三个方面的问题：

- 针对某类目上钻或下钻之前，必须知道其所属的类目层级，然后才能决定限制哪一级类目。如上述示例，限制一级类目 ID 等于 21。
- 假设分三级类目统计最近一天的 GMV，由于某些叶子类目直接是一级类目或二级类目（比如类目 ID 等于 121456022 的类目，其是叶子类目），和交易事实表关联之后，其对应的三级类目为空，导致根据三级类目统计最近一天的 GMV 时，类目 ID 等于 121456022 的交易无法被统计到。下游数据统计时，为了规避此问题，如果此类目对应的三级类目为空，则取二级类目；如果二级类目仍为空，则取一级类目。

所以针对此问题，下游数据统计时，类目层次结构扁平化的另一种方式是回填，将类目向下虚拟，具体数据存储示例如表 10.13 所示，其中粗体部分为回填内容。阿里巴巴中文站的类目体系使用此种方式。

表 10.13 数据存储示例 2

类目 ID	类目级别	是否叶子	一级类目	二级类目	三级类目
21	1	N	21	<b>21</b>	<b>21</b>
50026576	2	N	21	50026576	<b>50026576</b>
50026579	3	Y	21	50026576	<b>50026579</b>
121456022	2	Y	21	121456022	<b>121456022</b>
...	...	...	...	...	...

- 扁平化仅包含固定数量的级别，对于非平衡层次结构，可以通过预留级别的方式来解决，但扩展性较差。

## 2. 层次桥接表

针对层次结构扁平化所存在的问题，可以采用桥接表的方式来解决，不需要预先知道所属层级，不需要回填，也可解决非均衡层次结构的问题。与扁平化方法相比，该方法适合解决更宽泛的分析问题，灵活性好；但复杂性高，使用成本高。

仍然以类目为例，模型设计如图 10.5 所示。

对于上面提到的类目，使用树形结构表示如图 10.6 所示。

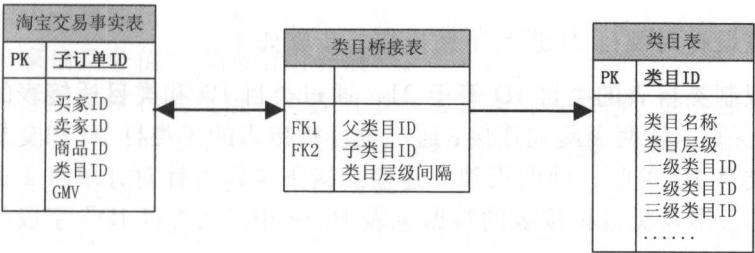


图 10.5 类目

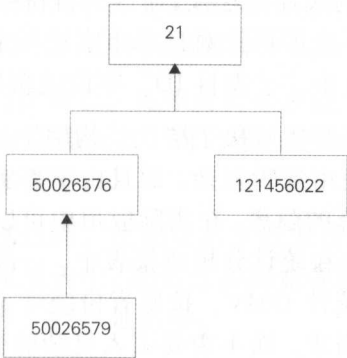


图 10.6 类目树

针对此类目树，类目桥接表的内容如表 10.14 所示。

表 10.14 类目桥接表

父类目 ID	子类目 ID	类目层级间隔
21	21	0
21	50026576	1
21	121456022	1
21	50026579	2
50026576	50026576	0
50026576	50026579	1
121456022	121456022	0

假设针对类目 21 进行下钻操作，步骤如下：

限制类目表的类目 ID 等于 21，通过类目 ID 和类目桥接表的父类目 ID 关联，使两表建立连接；通过类目桥接表的子类目 ID 和交易事实表的类目 ID 关联，使两表建立连接；接下来即可针对订单事实进行下钻操作。涉及类目桥接表的数据见表 10.14 中“父类目 ID”字段的粗体部分（21）。

假设针对类目 50026579 进行上钻操作，步骤如下：

限制类目表的类目 ID 等于 50026579，通过类目 ID 和类目桥接表的子类目 ID 关联，使两表建立连接；通过类目桥接表的父类目 ID 关联，使两表建立连接；接下来即可针对订单事实进行上钻操作。涉及类目桥接表的数据见表 10.14 中“子类目 ID”字段的粗体部分（50026579）。

可以看到，层次桥接表解决了层次结构扁平化带来的一些问题；但是其加工逻辑复杂，使用逻辑复杂，而且由于事实表和桥接表的多对多关系而带来了双重计算的隐患。在实际应用中可以根据具体的业务需求来选择技术方案，比如在统计分析或报表中，一般都是按照固定级别进行，如按照一级类目统计 GMV、按照省份统计 GMV 等，现在的扁平化设计完全可以满足需求，而不需要引入复杂的桥接表。很多时候，简单、直接的技术方案却是最好的解决方案。

## 10.4.2 行为维度

在阿里巴巴的数据仓库中，存在很多维表，如卖家主营类目维度、卖家主营品牌维度、用户常用地址维度等。其中卖家主营类目和主营品牌通过卖家的商品分布和交易分布情况，采用算法计算得到；卖家常用地址通过最近一段时间内物流中卖家的发货地址和买家的收货地址进行统计得到。类似的维度，都和事实相关，如交易、物流等，称之为“行为维度”，或“事实衍生的维度”。

按照加工方式，行为维度可以划分为以下几种：

- 另一个维度的过去行为，如买家最近一次访问淘宝的时间、买家最近一次发生淘宝交易的时间等。

- 快照事实行为维度，如买家从年初截至当前的淘宝交易金额、买家信用分值、卖家信用分值等。
- 分组事实行为维度，将数值型事实转换为枚举值。如买家从年初截至当前的淘宝交易金额按照金额划分的等级、买家信用分值按照分数划分得到的信用等级等。
- 复杂逻辑事实行为维度，通过复杂算法加工或多个事实综合加工得到。如前面提到的卖家主营类目，商品热度根据访问、收藏、加入购物车、交易等情况综合计算得到。

对于行为维度，有两种处理方式，其中一种是将其冗余至现有的维表中，如将卖家信用等级冗余至卖家维表中；另一种是加工成单独的行为维表，如卖家主营类目。具体采用哪种方式主要参考如下两个原则：

第一，避免维度过快增长。比如对商品表进行了极限存储，如果将商品热度加入现有的商品维表中，则可能会使每日商品变更占比过高，从而导致极限存储效果较差。

第二，避免耦合度过高。比如卖家主营类目，加工逻辑异常复杂，如果融合进现有的卖家维表中，那么过多的业务耦合会导致卖家维表刷新逻辑复杂、维护性差、产出延迟等。

### 10.4.3 多值维度

对于多值维度，一种情况是事实表的一条记录在某维表中有多条记录与之对应。比如对于淘宝交易订单，买家一次购买了多种商品，如一件毛衣和两双袜子，称为交易父订单；对于每种商品的交易，称为交易子订单；此交易父订单有两个子订单与之对应。假设设计交易父订单事实表，则对于此事实表的每一条记录，在商品表中都有一到多条记录与之对应。

针对多值维度，常见的处理方式有三种，可以根据业务的表现形式和统计分析需求进行选择。

第一种处理方式是降低事实表的粒度。在淘宝交易中，前台业务和商业智能关注交易子订单，所以在数据仓库模型设计中，将交易订单设

计为子订单粒度，对于每个子订单，只有一种商品与之对应。对于其中的事实，则采用分摊到子订单的方式来解决。但很多时候，事实表的粒度是不能降低的，多值维度的出现是无法避免的。

第二种处理方式是采用多字段。比如在房地产销售中，每次合同签订都可能存在多个买受方的情况，如夫妻合买等。对于合同签订事实表，每条记录可能对应多个买受方，而合同已经是此事实中的最细粒度，无法通过降低粒度的方式来解决。由于合同签订的要受人一般不会太多，所以一般采用多字段方式。考虑到扩展性，可以通过预留字段的方式，如超过三个买受方时，其余买受方填写至“其他买受方”字段。模型设计如图 10.7 所示。

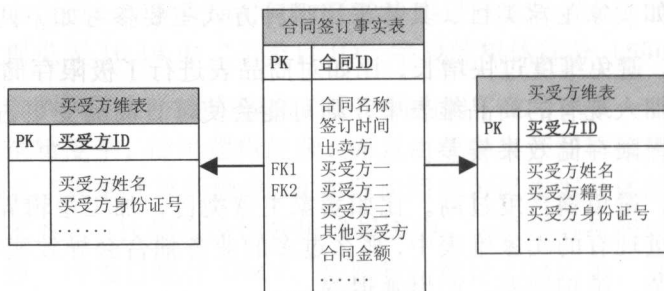


图 10.7 采用多字段方式处理多值维度

第三种处理方式是采用较为通用的桥接表。桥接表方式更加灵活、扩展性更好，但逻辑复杂、开发和维护成本较高，可能带来双重计算的风险，选择此方式需慎重。通过在事实表和维表之间开发一个分组表，通过此分组表建立连接。模型设计如图 10.8 所示，其中桥接表包含和事实表关联的分组 KEY，以及作为买受方维表外键的买受方 ID。如果事实表的一条记录对应两个买受方，则桥接表针对这两个买受方建立两条记录，分组 KEY 相同。

假设根据买受方籍贯统计 2015 年的合同总金额，如果某合同有两个买受方，籍贯分别是浙江和山东，那么此合同总金额将会分别统计浙江和山东的，造成双重计算。双重计算不一定是错误，对于一些业务需求是合理的；但对于另一些业务需求，则需要规避。

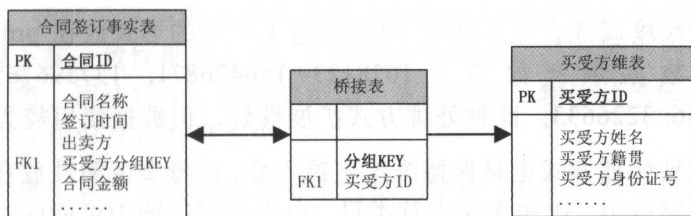


图 10.8 采用桥接表方式多理多值维度

### 10.4.4 多值属性

维表中的某个属性字段同时有多个值，称之为“多值属性”。它是多值维度的另一种表现形式。在阿里巴巴的数据仓库中，存在很多维表，如商品 SKU 维表、商品属性维表、商品标签维表等。每个商品均有一到多个 SKU、一到多个属性和一到多个标签，所以商品和 SKU、属性、标签都是多对多的关系。淘宝商品 SKU 和属性信息示例如图 10.9 所示。

尺码

L/90 (时尚提花, 纯棉保证)


XL/95 (送妈妈的贴心礼物)

XXL/100 (舒适透气亲肤)

XXXL/105 (把爱带回家)


4XL【建议145-160斤】

颜色分类










<p>品牌: 乙王</p> <p>颜色分类: 紫红中领 咖啡色中领 紫...</p> <p>材质: 棉</p> <p>面料: 纯棉</p> <p>款号: 5039</p> <p>功能: 保暖</p> <p>是否加绒: 加绒</p>	<p>袖长: 长袖</p> <p>裤长: 长裤</p> <p>成分含量: 95%以上</p> <p>图案: 其他</p> <p>尺码: L/90 (时尚提花, 纯棉保证) ...</p> <p>领型: 高领</p> <p>是否贴片: 无贴片</p>	<p>适用对象: 中年</p> <p>厚薄: 加厚</p> <p>克重: 401g(含)-450g(含)</p> <p>服装款式细节: 提花</p> <p>适用性别: 女</p> <p>层数: 单层</p>
---	--	---

图 10.9 淘宝商品 SKU 和属性信息示例

对于多值属性，常见的处理方式有三种，可以根据具体情况进行选择。

第一种处理方式是保持维度主键不变，将多值属性放在维度的一个属性字段中。比如对于商品属性（注：此属性是业务上的含义，和维度



建模中的维度属性含义不同)，可以通过 k-v 对的形式放在 property 字段中，数据示例如下：10281239:156426871; 137396765:29229; 137400766: 3226633。此种处理方式扩展性好，但数据使用较为麻烦。

第二种处理方式也是保持维度主键不变，但将多值属性放在维度的多个属性字段中。比如卖家主营类目，由于卖家店铺中可能同时会销售男装、女装、内衣等，所以卖家主营类目可能有多个，但业务需求是只取根据算法计算得到的 TOP3。针对此种情况，维度的多值属性字段具体值的数量固定，可以采用多个属性字段进行存储，方便数据统计分析和报表展示。如果多值属性字段具体值的数量不固定，则可以采用预留字段的方式，但扩展性较差。卖家主营类目维度设计如图 10.10 所示。

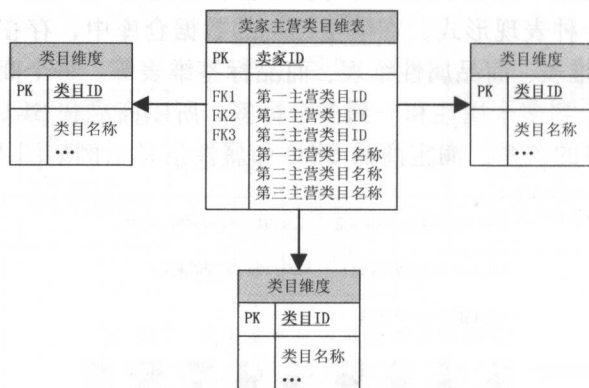


图 10.10 卖家主营类目维度设计

第三种处理方式是维度主键发生变化，一个维度值存放多条记录。比如商品 SKU 维表，对于每个商品，有多少 SKU，就有多少记录，主键是商品的 ID 和 SKU 的 ID。此种处理方式扩展性好，使用方便，但需要考虑数据的急剧膨胀情况。比如淘宝商品属性表采用了此种处理方式，数据记录达到几百亿的级别。

#### 10.4.5 杂项维度

在维度建模中，有一种维度叫 Junk Dimension，中文一般翻译为“杂

项维度”。杂项维度是由操作型系统中的指示符或者标志字段组合而成的，一般不在一致性维度之列。比如淘宝交易订单的交易类型字段，包括话费充值、司法拍卖、航旅等类型；支付状态、物流状态等，它们在源系统中直接保存在交易表中。

一个事实表中可能会存在多个类似的字段，如果作为事实存放在事实表中，则会导致事实表占用空间过大；如果单独建立维表，外键关联到事实表，则会出现维度过多的情况；如果将这些字段删除，则会有人不同意。

这时，通常的解决方案就是建立杂项维度，将这些字段建立到一个维表中，在事实表中只需保存一个外键即可。多个字段的不同取值组成一条记录，生成代理键，存入维表中，并将该代理键保存到相应的事实表字段下。建议不要直接使用所有的组合生成完整的杂项维表，在抽取遇到新的组合时生成相应的记录即可。杂项维度的 ETL 过程比一般的维度略微复杂些。

但在阿里巴巴的实践中，杂项维度不仅包含上述指示符、状态或分类等枚举字段，还包含很多非枚举字段，如交易留言、交易属性（由若干 k-v 对组成）、交易标签（由二进制位表示）等。针对这些字段，不可能生成所有的组合；同时，由于在分布式计算系统中生成代理键的复杂度，一般在逻辑建模中，会使用实体的主键作为杂项维度的主键。只考虑杂项维度，忽略其他维度，如图 10.11 所示。

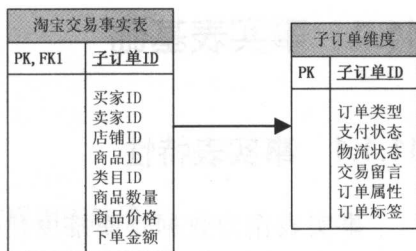


图 10.11 杂项维度

但子订单维度一般是逻辑模型，物理实现时不进行物理化，订单杂项维度和其他维度一起，会将维度属性退化至事实表中，详情在事实表中描述。

注：本章节部分理论来自于 Christopher Adamson 的 Star Schema -The Complete Reference 和 Ralph Kimball 的 The Data Warehouse Toolkit-The Definitive Guide to Dimensional Modeling。本书结合阿里的实践进行讲解，细节内容请参考各自著作进行学习。

# 第11章

## 事实表设计

### 11.1 事实表基础

#### 11.1.1 事实表特性

事实表作为数据仓库维度建模的核心，紧紧围绕着业务过程来设计，通过获取描述业务过程的度量来表达业务过程，包含了引用的维度和与业务过程有关的度量。

事实表中一条记录所表达的业务细程度被称为粒度。通常粒度可以通过两种方式来表述：一种是维度属性组合所表示的细节程度；一种是所表示的具体业务含义。

作为度量业务过程的事实，一般为整型或浮点型的十进制数值，有可加性、半可加性和不可加性三种类型。可加性事实是指可以按照与事实表关联的任意维度进行汇总。半可加性事实只能按照特定维度汇总，不能对所有维度汇总，比如库存可以按照地点和商品进行汇总，而按时

间维度把一年中每个月的库存累加起来则毫无意义。还有一种度量完全不具备可加性,比如比率型事实。对于不可加性事实可分解为可加的组件来实现聚集。

相对维表来说,通常事实表要细长得多,行的增加速度也比维表快很多。

维度属性也可以存储到事实表中,这种存储到事实表中的维度列被称为“退化维度”。与其他存储在维表中的维度一样,退化维度也可以用来进行事实表的过滤查询、实现聚合操作等。

事实表有三种类型:事务事实表、周期快照事实表和累积快照事实表,具体内容后面章节会详细介绍。事务事实表用来描述业务过程,跟踪空间或时间上某点的度量事件,保存的是最原子的数据,也称为“原子事实表”。周期快照事实表以具有规律性的、可预见的时间间隔记录事实,时间间隔如每天、每月、每年等。累积快照事实表用来表述过程开始和结束之间的关键步骤事件,覆盖过程的整个生命周期,通常具有多个日期字段来记录关键时间点,当过程随着生命周期不断变化时,记录也会随着过程的变化而被修改。

### 11.1.2 事实表设计原则

#### 原则 1: 尽可能包含所有与业务过程相关的事实

事实表设计的目的是为了度量业务过程,所以分析哪些事实与业务过程有关是设计中非常重要的关注点。在事实表中应该尽量包含所有与业务过程相关的事实,即使存在冗余,但是因为事实通常为数字型,带来的存储开销也不会很大。

#### 原则 2: 只选择与业务过程相关的事实

在选择事实时,应该注意只选择与业务过程相关的事实。比如在订单的下单这个业务过程的事实表设计中,不应该存在支付金额这个表示支付业务过程的事实。

### 原则 3：分解不可加性事实为可加的组件

对于不具备可加性条件的事实，需要分解为可加的组件。比如订单的优惠率，应该分解为订单原价金额与订单优惠金额两个事实存储在事实表中。

### 原则 4：在选择维度和事实之前必须先声明粒度

粒度的声明是事实表设计中不可忽视的重要一步，粒度用于确定事实表中一行所表示业务的细节层次，决定了维度模型的扩展性，在选择维度和事实之前必须先声明粒度，且每个维度和事实必须与所定义的粒度保持一致。在设计事实表的过程中，粒度定义得越细越好，建议从最低级别的原子粒度开始，因为原子粒度提供了最大限度的灵活性，可以支持无法预期的各种细节层次的用户需求。在事实表中，通常通过业务描述来表述粒度，但对于聚集性事实表的粒度描述，可采用维度或维度属性组合的方式。

### 原则 5：在同一个事实表中不能有多种不同粒度的事实

事实表中的所有事实需要与表定义的粒度保持一致，在同一个事实表中不能有多种不同粒度的事实。

如表 11.1 所示为机票支付成功事务事实表，粒度为票一级的，而在实际业务中，一个订单可以同时支付多张票，如 ID 为 100901 的订单包含三张机票，ID 为 100902 的订单包含两张机票，ID 为 100903 的订单包含一张机票。在该事实表的设计中，票支付金额和票折扣金额两个事实与表定义的粒度一致，并且支持按表的任意维度汇总，可以添加进该事实表中。而订单支付金额和订单票数作为上一层粒度的订单级事实，与该票级事实表的粒度不一致，且不能进行汇总。比如订单 ID 为 100901 的订单支付金额为 3700 元，订单票数为 3 张，如果这两个度量在该表进行汇总计算总订单金额和总票数，则会造成重复计算的问题，所以不能作为该表的度量选入。

表 11.1 事务事实表

机票 ID	订单 ID	票支付金额	票折扣金额	订单支付金额	订单票数
23459801	100901	1000.00	100.00	3700.00	3
23459802	100901	1200.00	120.00	3700.00	3
23459803	100901	1500.00	150.00	3700.00	3
23459804	100902	1600.00	160.00	2600.00	2
23459805	100902	1000.00	100.00	2600.00	2
23459806	100903	1500.00	150.00	1500.00	1
.....					

原则 6：事实的单位要保持一致

对于同一个事实表中事实的单位，应该保持一致。比如原订单金额、订单优惠金额、订单运费金额这三个事实，应该采用一致的计量单位，统一为元或分，以方便使用。

原则 7：对事实的 null 值要处理

对于事实表中事实度量为 null 值的处理，因为在数据库中 null 值对常用数字型字段的 SQL 过滤条件都不生效，比如大于、小于、等于、大于或等于、小于或等于，建议用零值填充。

原则 8：使用退化维度提高事实表的易用性

在 Kimball 的维度建模中，通常按照星形模型的方式来设计，对于维度的获取采用的是通过事实表的外键关联专门的维表的方式，谨慎使用退化维度。而在大数据领域的事实表设计中，则大量采用退化维度的方式，在事实表中存储各种类型的常用维度信息。这样设计的目的是为了减少下游用户使用关联多个表的操作，直接通过退化维度实现对事实表的过滤查询、控制聚合层次、排序数据以及定义主从关系等。通过增加冗余存储的方式减少计算开销，提高使用效率。

11.1.3 事实表设计方法

在 Kimball 所著的 *The Data Warehouse Toolkit-The Definitive Guide*

toDimensional Modeling 一书中，对于维度模型设计采用四步设计方法：选择业务过程、声明粒度、确定维度、确定事实。

在当前的互联网大数据环境下，面对复杂的业务场景，为了更有效、准确地进行维度模型建设，基于 Kimball 的四步维度建模方法，我们进行了更进一步的改进。

### 第一步：选择业务过程及确定事实表类型。

在明确了业务需求以后，接下来需要进行详细的需求分析，对业务的整个生命周期进行分析，明确关键的业务步骤，从而选择与需求有关的业务过程。

以淘宝的正向订单流转为例，如图 11.1 所示。

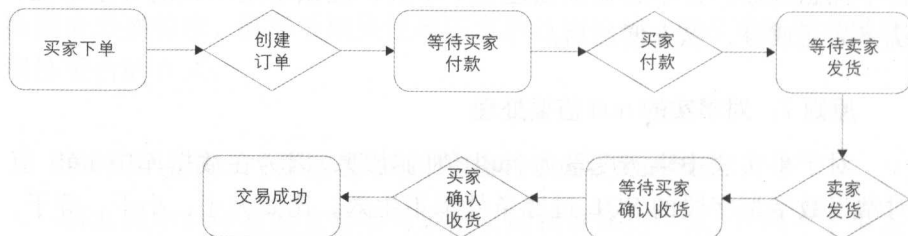


图 11.1 淘宝交易订单的流转过程

业务过程通常使用行为动词表示业务执行的活动。比如图 11.1 中的淘宝订单流转的业务过程有四个：创建订单、买家付款、卖家发货、买家确认收货。在明确了流程所包含的业务过程后，需要根据具体的业务需求来选择与维度建模有关的业务过程。比如是选择买家付款这个业务过程，还是选择创建订单和买家付款这两个业务过程，具体根据业务情况来确定。

在选择了业务过程以后，相应的事实表类型也随之确定了。比如选择买家付款这个业务过程，那么事实表应为只包含买家付款这一个业务过程的单事务事实表；如果选择的是所有四个业务过程，并且需要分析各个业务过程之间的时间间隔，那么所建立的事实表应为包含了所有四个业务过程的累积快照事实表。

## 第二步：声明粒度。

粒度的声明是事实表建模非常重要的一步，意味着精确定义事实表的每一行所表示的业务含义，粒度传递的是与事实表度量有关的细节层次。明确的粒度能确保对事实表中行的意思的理解不会产生混淆，保证所有的事实按照同样的细节层次记录。

应该尽量选择最细级别的原子粒度，以确保事实表的应用具有最大的灵活性。同时对于订单过程而言，粒度可以被定义为最细的订单级别。比如在淘宝订单中有父子订单的概念，即一个子订单对应一种商品，如果拍下了多种商品，则每种商品对应一个子订单；这些子订单一同结算的话，则会生成一个父订单。那么在这个例子中，事实表的粒度应该选择为子订单级别。

## 第三步：确定维度。

完成粒度声明以后，也就意味着确定了主键，对应的维度组合以及相关的维度字段就可以确定了，应该选择能够描述清楚业务过程所处的环境的维度信息。比如在淘宝订单付款事务事实表中，粒度为子订单，相关的维度有买家、卖家、商品、收货人信息、业务类型、订单时间等维度。

## 第四步：确定事实。

事实可以通过回答“过程的度量是什么”来确定。应该选择与业务过程有关的所有事实，且事实的粒度要与所声明的事实表的粒度一致。事实有可加性、半可加性、非可加性三种类型，需要将不可加性事实分解为可加的组件。

比如在淘宝订单付款事务事实表中，同粒度的事实有子订单分摊的支付金额、邮费、优惠金额等。

## 第五步：冗余维度。

在传统的维度建模的星形模型中，对维度的处理是需要单独存放在



专门的维表中的,通过事实表的外键获取维度。这样做的目的是为了减少事实表的维度冗余,从而减少存储消耗。而在大数据的事实表模型设计中,考虑更多的是提高下游用户的使用效率,降低数据获取的复杂性,减少关联的表数量。所以通常事实表中会冗余方便下游用户使用的常用维度,以实现事实表的过滤查询、控制聚合层次、排序数据以及定义主从关系等操作。

比如在淘宝订单付款事务事实表中,通常会冗余大量的常用维度字段,以及商品类目、卖家店铺等维度信息。

## 11.2 事务事实表

订单作为交易行为的核心载体,直观反映了交易的状况。订单的流转会产生很多业务过程,而下单、支付和成功完结三个业务过程是整个订单的关键节点。获取这三个业务过程的笔数、金额以及转化率是日常数据统计分析的重点,事务事实表设计可以很好地满足这个需求。本节将介绍三种不同事务事实表的设计方式,以及在淘宝交易订单中关于邮费和折扣分摊到子订单的算法。

### 11.2.1 设计过程

任何类型的事件都可以被理解作为一种事务。比如交易过程中的创建订单、买家付款,物流过程中的揽货、发货、签收,退款中的申请退款、申请小二介入等,都可以被理解作为一种事务。事务事实表,即针对这些过程构建的一类事实表,用以跟踪定义业务过程的个体行为,提供丰富的分析能力,作为数据仓库原子的明细数据。下面以淘宝交易事务事实表为例,阐述事务事实表的一般设计过程。

#### (1) 选择业务过程

图 11.1 给出了淘宝交易订单的流转过程,其中介绍了四个重要过

程：创建订单、买家付款、卖家发货、买家确认收货，即下单、支付、发货和成功完结四个业务过程。这四个业务过程不仅是交易过程中的重要时间节点，而且也是下游统计分析的重点，因此淘宝交易事务事实表设计着重从这四个业务过程进行展开。

Kimball 维度建模理论认为，为了便于进行独立的分析研究，应该为每个业务过程建立一个事实表。对于是否将不同业务过程放到同一个事实表中，将在下一节中详细介绍。

## (2) 确定粒度

业务过程选定以后，就要针对每个业务过程确定一个粒度，即确定事务事实表每一行所表达的细节层次。下面先介绍淘宝订单的产生过程。

淘宝出售商品主要分两类卖家：一类是个人性质的闲置卖家，主要出售闲置的或者二手商品；一类是拥有店铺的卖家，以出售新商品为主。接下来主要以店铺类交易订单为例进行介绍。在淘宝下单交易时，有两种方式：一种是选定商品后直接购买，这样会产生一个交易订单；一种是将多种商品加入到购物车中，然后一起结算，此时对于每一种商品都会产生一个订单，同时对于同一个店铺会额外产生一个订单，即父订单；由于是在同一个店铺购买的，所以父订单会承载订单物流、店铺优惠等信息。而对于每一种商品产生的订单就称为子订单，子订单记录了父订单的订单号，并且有子订单标志。如果在同一个店铺只购买了一种商品，则会将父子订单进行合并，只保留一条订单记录。如图 11.2 和图 11.3 所示示例。

了解了淘宝交易订单的产生过程后，现在为淘宝交易事务事实表确定粒度。如第 1 步所述，在淘宝交易过程中有四个重要业务过程，需要为每个业务过程确定一个粒度。其中下单、支付和成功完结三个业务过程选择交易子订单粒度，即每个子订单为事务事实表的一行，每个子订单所表达的细节信息为：交易时间、卖家、买家、商品，即选择图 11.2 和图 11.3 中订单 ID 为 1、4、5、6、7、8、9 的子订单作为事务事实表的每一行。卖家发货这个业务过程可以选择子订单粒度，即将每个子订单作为卖家发货事实表的一个细节。然而，在实际操作中发现，卖家发货更多的是物流单粒度而非子订单粒度，同一个子订单可以拆分成多个物流单进行发货。在事务事实表设计过程中，秉承确定为最细粒度的原

则，因此对于卖家发货确定为物流单粒度，和其他三个业务过程不同，这样可以更好地给下游统计分析带来灵活性。

订单ID	父订单ID	创建时间	买家ID	卖家ID	商品ID	金额	数量	是否子订单	是否父订单
1	1	2016-01-01 00:00:00	111	222	11111	100.0	1	Y	Y
9	9	2016-01-03 00:00:00	444	555	66666	500.0	2	Y	Y

图 11.2 父子订单合并成一条记录

订单ID	父订单ID	创建时间	买家ID	卖家ID	商品ID	金额	数量	是否子订单	是否父订单
2	0	2016-01-01 00:00:00	111	222	0	0.0	3	N	Y
4	2	2016-01-01 00:00:00	111	222	11111	100.0	1	Y	N
5	2	2016-01-01 00:00:00	111	222	22222	50.0	2	Y	N
3	0	2016-01-02 00:00:00	333	777	0	0.0	6	N	Y
6	3	2016-01-02 00:00:00	333	777	33333	70.0	1	Y	N
7	3	2016-01-02 00:00:00	333	777	44444	80.0	2	Y	N
8	3	2016-01-02 00:00:00	333	777	55555	90.0	3	Y	N

图 11.3 父子订单拆开成多条记录

(3) 确定维度

选定好业务过程并且确定粒度后，就可以确定维度信息了。在淘宝交易事务事实表设计过程中，按照经常用于统计分析的场景，确定维度包含：买家、卖家、商品、商品类目、发货地区、收货地区、父订单维度以及杂项维度。由于订单的属性较多，比如订单的业务类型、是否无线交易、订单的 attributes 属性等，对于这些使用较多却又无法归属到上述买卖家或商品维度中的属性，则新建一个杂项维度进行存放，如图 11.4 所示。

(4) 确定事实

作为过程度量的核心，事实表应该包含与其描述过程有关的所有事实。以淘宝交易事务事实表为例，选定三个业务过程——下单、支付和成功完结，不同的业务过程拥有不同的事实。比如在下单业务过程中，需要包含下单金额、下单数量、下单分摊金额；在支付业务过程中，包含支付金额、分摊邮费、折扣金额、红包金额、积分金额；在完结业务过程中包含确认收货金额等。由于粒度是子订单，所以对于一些父订单上的金额需要分摊到子订单上，比如父订单邮费、父订单折扣等。具体的分摊算法将在“父子事实的处理方式”一节中介绍。

根据 Kimball 维度建模理论，经过以上四步，淘宝交易事务事实表

已成型，可以满足下游分析统计的需要。然而，阿里巴巴数据仓库在建模时，基于以上四步增加了一步——退化维度，这个过程在 Kimball 维度建模中也有所提及；但阿里巴巴数据仓库出于效率和资源的考虑，将常用维度全部退化到事实表中，使下游分析使用模型更加方便。

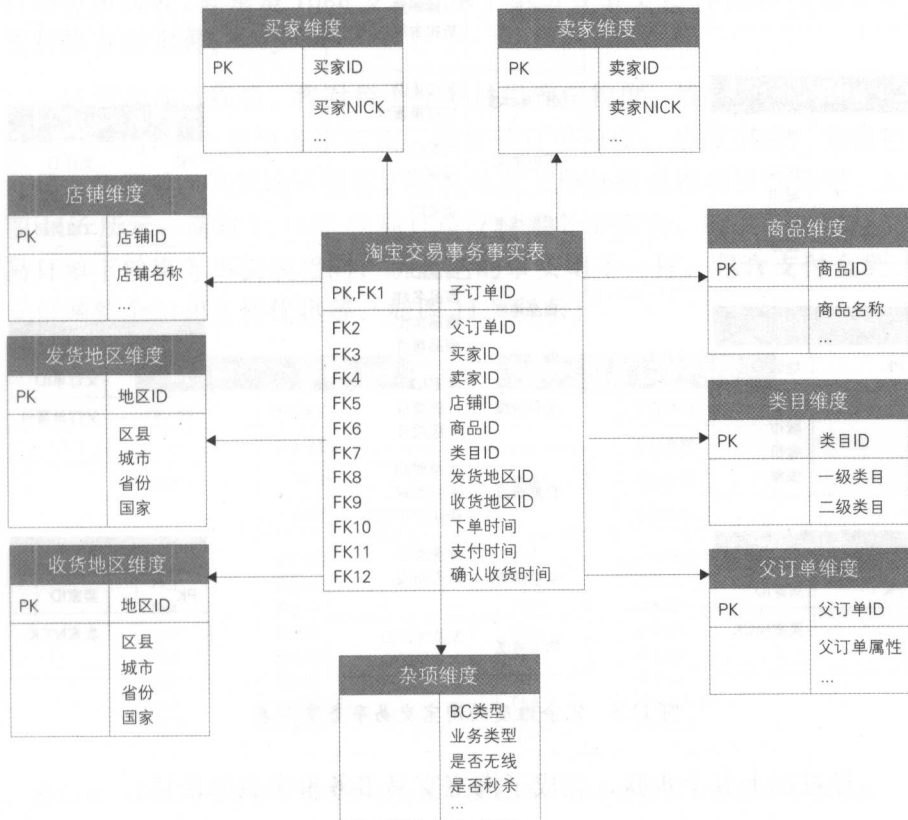


图 11.4 淘宝交易事务事实表（确定维度）

### （5）冗余维度

在确定维度时，包含了买卖家维度、商品维度、类目维度、收发货维度等，Kimball 维度建模理论建议在事实表中只保存这些维表的外键，而淘宝交易事务事实表在 Kimball 维度建模基础之上做了进一步的优化，将买卖家星级、标签、店铺名称、商品类型、商品特征、商品属性、类目层级等维度属性都冗余到事实表中，提高对事实表进行过滤查询、

统计聚合的效率，如图 11.5 所示。

店铺维度		淘宝交易事务事实表		商品维度	
PK	店铺ID	PK	子订单ID	PK	商品ID
	店铺名称 ...	度量	支付金额 分摊邮费 折扣金额 ...		商品名称 ...
发货地区维度		父订单维度	父订单ID 父订单属性	类目维度	
PK	地区ID	买家维度	买家ID 买家Nick	PK	类目ID
	区县 城市 省份 国家	卖家维度	卖家ID 卖家Nick		一级类目 二级类目
收货地区维度		商品维度	商品ID 商品名称 商品类型 商品属性	父订单维度	
PK	地区ID	类目维度	类目ID 一级类目 二级类目	PK	父订单ID
	区县 城市 省份 国家	日期维度	下单时间 支付时间 确认收货时间		父订单属性 ...
买家维度		杂项维度	业务类型 是否无线 ...	卖家维度	
PK	买家ID	物流维度	发货地区ID 收货地区ID	PK	卖家ID
	买家NICK ...				卖家NICK ...

图 11.5 冗余维度的淘宝交易事务事实表

经过以上五个步骤，完成了淘宝交易事务事实表的设计。

但在设计过程中遗留一个问题，即对于单一事实表中是否包含多个业务过程，还没有给出定论。接下来将通过在淘宝和 1688 交易过程中采用不同的设计方案来阐述两种设计方法。

11.2.2 单事务事实表

单事务事实表，顾名思义，即针对每个业务过程设计一个事实表。这样设计的优点不言而喻，可以方便地对每个业务过程进行独立的分析

研究。1688 交易流程则采用这种模式构建事务事实表。

1688 交易和淘宝交易相似，主要流程也是下单、支付、发货和完结，而在这四个关键流程中 1688 交易选择下单和支付两个业务过程设计事务事实表，分别是 1688 交易订单下单事务事实表和 1688 交易订单支付事务事实表。

选定业务过程后，将对每个业务过程确定粒度、维度和事实。对于 1688 交易订单下单事务事实表，确定子订单粒度，选择买家、卖家、商品、父订单、收货地区维度，事实包含下单分摊金额和折扣金额，如图 11.6 所示；而对于 1688 交易订单支付事务事实表，粒度和维度与交易订单下单事务事实表相同，所表达的事实则不一样，包含支付金额、支付调整金额和支付优惠等，如图 11.7 所示。

1688交易订单下单事务事实表		1688交易订单支付事务事实表	
PK,FK1	子订单ID	PK,FK1	子订单ID
度量	下单全额	度量	支付全额
	下单优惠		支付优惠
	下单折扣		支付折扣
FK2	父订单ID	FK2	父订单ID
FK3	买家ID	FK3	买家ID
FK4	卖家ID	FK4	卖家ID
FK5	店铺ID	FK5	店铺ID
FK6	商品ID	FK6	商品ID
FK7	类目ID	FK7	类目ID
FK9	收货地区ID	FK9	收货地区ID
FK10	下单时间	FK10	支付时间

图 11.6 1688 交易订单下单事务事实表    图 11.7 1688 交易订单支付事务事实表

1688 交易针对下单和支付分别建立单事务事实表后，每天的下单记录则进入当天的下单事务事实表中，每天的支付记录进入当天的支付事务事实表中，由于事实表具有稀疏性质，因此只有当天数据才会进入当天的事实表中。下面以具体交易订单为例，展示单事务事实表的设计实例。如图 11.8 所示，order1 在 2016-01-01 下单并且在当天完成支付；order2 和 order3 在 2016-01-01 下单并且在 2016-01-02 完成支付。如图 11.9 和图 11.10 所示，order1、order2 和 order3 写入下单事务事实表中，业务日期（下单日期）均为 2016-01-01；order1、order2 和 order3 也分

别写入支付事务事实表中，业务日期（支付日期）分别为 2016-01-01、2016-01-02 和 2016-01-02。

订单ID	父订单ID	下单时间	支付时间	完结时间	买家ID	卖家ID	商品ID	下单度量	支付度量	是否子订单	是否父订单
order1	mord1	2016-01-01 08:00:00	2016-01-01 10:00:00	2016-01-02 10:00:00	111	222	aa	...	...	Y	Y
order2	mord2	2016-01-01 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	0	...	...	N	Y
order3	mord2	2016-01-01 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	bb	...	...	Y	N
order4	mord2	2016-01-01 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	cc	...	...	Y	N

图 11.8 1688 交易订单详情实例

业务日期	订单ID	父订单ID	下单时间	买家ID	卖家ID	商品ID	下单度量
20160101	order1	mord1	2016-01-01 08:00:00	111	222	aa	...
20160101	order3	mord2	2016-01-01 09:00:00	333	444	bb	...
20160101	order4	mord2	2016-01-01 09:00:00	333	444	cc	...

图 11.9 1688 交易订单下单事务事实表数据实例

业务日期	订单ID	父订单ID	支付时间	买家ID	卖家ID	商品ID	支付度量
20160101	order1	mord1	2016-01-01 08:00:00	111	222	aa	...
20160102	order3	mord2	2016-01-01 09:00:00	333	444	bb	...
20160102	order4	mord2	2016-01-01 09:00:00	333	444	cc	...

图 11.10 1688 交易订单支付事务事实表数据实例

11.2.3 多事务事实表

多事务事实表，将不同的事实放到同一个事实表中，即同一个事实表包含不同的业务过程。多事务事实表在设计时有两种方法进行事实的处理：①不同业务过程的事实使用不同的事实字段进行存放；②不同业务过程的事实使用同一个事实字段进行存放，但增加一个业务过程标



签。接下来将通过淘宝交易事务事实表和淘宝收藏商品事务事实表分别阐述其设计方法。

### 1. 淘宝交易事务事实表

淘宝交易事务事实表采取将不同业务过程的事实使用不同事实字段进行存放的设计模式。淘宝交易事务事实表中同时包含了下单、支付和成功完结三个业务过程，这三个业务过程拥有相同的粒度，都是子订单粒度，也比较适合放到同一个事实表中。选择业务过程时没有把发货也加到此事务事实表中，原因是发货的粒度比子订单更细，属于不同粒度上的业务过程，因此没有放到同一个事实表中。

在确定好业务过程和粒度后，下一步就是确定维度和事实。对于不同的业务过程和粒度，一般而言，维度也不完全一致。但是在设计淘宝交易事务事实表时，根据分析统计，常用维度比较一致，因此在维度层面可以保证这三个业务过程放到同一个事务事实表中。这里的维度也是在交易过程中比较常见的，如包括买家、卖家、商品、类目、店铺、收发货地区等，无论在哪一个业务过程中，都需要按照这些维度进行统计分析。

将多个业务过程放到同一个事实表中，将要面对的是如何处理多个事实。淘宝交易事务事实表中包含了下单、支付和成功完结三个业务过程，则需要包含下单度量、支付度量和成功完结度量信息，这里的解决方案是针对每个度量都使用一个字段进行保存，即不同的事实使用不同的字段进行存放；如果不是当前业务过程的度量，则采取零值处理方式。比如在下单业务过程中，对于支付度量和成功完结度量全部置为 0，其他业务过程类似处理。

同一个事实表中包含了多个业务过程，在表中如何进行标记呢？淘宝交易事务事实表采取了这样的解决方案，即针对每个业务过程打一个标签，标记当天是否是这个业务过程，比如针对下单，则打一个是否当天下单的标签；针对支付，打一个是否当天支付的标签；针对成功完结，打一个是否当天成功完结的标签，标签之间互不相干。淘宝交易事务事实表如图 11.11 所示。

同样以具体交易订单为例，展示多事务事实表的设计实例，如图 11.12 所示，order1 在 2016-01-01 下单并且在当天完成支付；order2 和 order3 在 2016-01-01 下单并且在 2016-01-02 完成支付，在 2016-01-04



成功完结。淘宝交易多事务事实表数据实例如图 11.13 所示，同一个事实表中包含有多个业务过程数据。

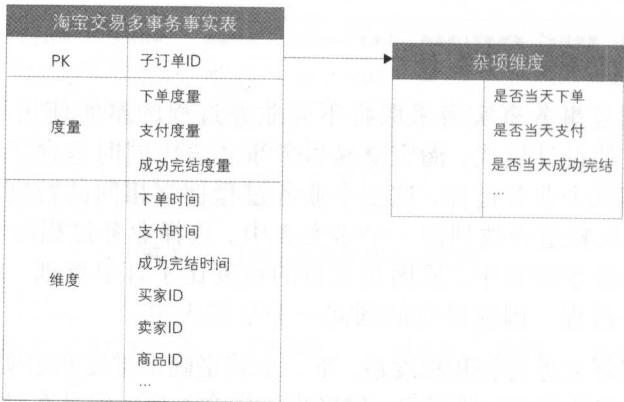


图 11.11 淘宝交易多事务事实表

订单ID	父订单ID	下单时间	支付时间	完结时间	买家ID	卖家ID	商品ID	下单度量	支付度量	是否子订单	是否父订单
order1	mord1	2016-01-01 08:00:00	2016-01-01 10:00:00	2016-01-02 10:00:00	111	222	aa	...	...	Y	Y
order2	mord2	2016-01-01 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	0	...	...	N	Y
order3	mord2	2016-01-01 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	bb	...	...	Y	N
order4	mord2	2016-01-01 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	cc	...	...	Y	N

图 11.12 淘宝交易订单详情实例

业务日期	订单ID	父订单ID	是否当天下单	是否当天支付	是否当天完结	下单时间	支付时间	完结时间	买家ID	卖家ID	商品ID	下单度量	支付度量	完结度量
20160101	order1	mord1	Y	Y	N	2016-01-01 08:00:00	2016-01-01 10:00:00	NULL	111	222	aa	...	...	0.0
20160101	order3	mord2	Y	N	N	2016-01-01 09:00:00	NULL	NULL	333	444	bb	...	0.0	0.0
20160101	order4	mord2	Y	N	N	2016-01-01 09:00:00	NULL	NULL	333	444	cc	...	0.0	0.0
20160102	order1	mord1	N	N	Y	2016-01-02 08:00:00	2016-01-02 10:00:00	2016-01-02 10:00:00	111	222	aa	...	...	...
20160102	order3	mord2	N	Y	N	2016-01-02 09:00:00	2016-01-02 09:00:00	NULL	333	444	bb	...	...	0.0
20160102	order4	mord2	N	Y	N	2016-01-02 09:00:00	2016-01-02 09:00:00	NULL	333	444	cc	...	...	0.0
20160104	order3	mord2	N	N	Y	2016-01-02 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	bb	...	...	...
20160104	order4	mord2	N	N	Y	2016-01-02 09:00:00	2016-01-02 09:00:00	2016-01-04 09:00:00	333	444	cc	...	...	...

图 11.13 淘宝交易多事务事实表数据实例

## 2. 淘宝收藏商品事务事实表

收藏和加购物车是淘宝购物过程中比较常见的两个行为,当用户遇到喜欢的商品或者店铺时可以选择收藏,然后下次继续浏览购买。这里以收藏事务事实表阐述多事务事实表在处理不同业务过程时使用同一个字段保存事实的设计方式。

收藏业务较为简单,商品和店铺的收藏业务相似,这里仅以收藏商品为例进行阐述。用户可以直接收藏一个商品,也可以删除所收藏的商品,所以在这个过程中包含了两个业务过程:收藏商品和删除商品。因此收藏商品事务事实表的第一步就是选择收藏商品和删除商品两个业务过程。业务过程确定后,接下来就是确定粒度。无论是收藏商品还是删除所收藏的商品,都是用户对商品的一个操作,因此这里确定为用户加上商品的粒度。

确定好业务过程和粒度后,接下来是确定维度和事实。由于粒度是用户加上商品,所以维度主要是用户维度和商品维度。为了使事实表信息更丰富,冗余了商品类目维度和商品所属卖家维度,收藏商品和删除商品业务过程所属的维度是一致的。

收藏商品和删除商品是两个不同的业务过程,但是确定了相同的粒度和维度,所以考虑设计多事务事实表,将这两个业务过程放到同一个事实表中,只是在不同业务过程的事实上进行区分。在前面的“淘宝交易事务事实表”中是使用不同字段存放不同业务过程的事实,这里的解决方案是使用同一个字段存放不同业务过程的事实,使用标签字段区分不同业务过程,比如收藏事务事实表使用一个“收藏事件类型”字段来区分是收藏商品还是删除商品。收藏商品和删除商品的事实主要是商品价格,不过收藏事务事实表更多的是无事实的事实表,一般用于统计收藏或者删除的次数。

下面通过实例来说明收藏商品事务事实表的设计过程,如图 11.14、图 11.15 所示。

收藏时间	删除时间	商品ID	会员ID	商品度量
2016-01-01 08:00:00	NULL	aa	111	...
2016-01-01 12:00:00	2016-01-01 18:00:00	aa	222	...
2016-01-01 08:00:00	2016-01-01 08:00:00	bb	333	...

图 11.14 收藏商品明细

业务日期	收藏时间	删除时间	商品ID	会员ID	商品度量	收藏删除类型
20160101	2016-01-01 08:00:00	NULL	aa	111	...	collect
20160101	2016-01-01 12:00:00	2016-01-01 18:00:00	aa	222	...	delete
20160101	2016-01-01 08:00:00	NULL	bb	333	...	collect
20160102	2016-01-01 08:00:00	2016-01-02 08:00:00	bb	333	...	delete

图 11.15 收藏商品事务事实表数据实例

### 3. 多事务事实表的选择

上面介绍了两种多事务事实表的设计方式，在实际应用中需要根据业务过程进行选择。由于是多事务事实表，因此在事实表中包含多个业务过程：

- 当不同业务过程的度量比较相似、差异不大时，可以采用第二种多事务事实表的设计方式，使用同一个字段来表示度量数据。但这种方式存在一个问题——在同一个周期内会存在多条记录。
- 当不同业务过程的度量差异较大时，可以选择第一种多事务事实表的设计方式，将不同业务过程的度量使用不同字段冗余到表中，非当前业务过程则置零表示。这种方式所存在的问题是度量字段零值较多。

## 11.2.4 两种事实表对比

前面介绍了单事务事实表和多事务事实表的设计过程，同时给出了关于 1688 和淘宝不同事务事实表的实例。目前两类事实表都有实际的

应用，但具体哪一种设计方式更优，我们接下来进行分析。

### 1. 业务过程

对于单事务事实表，一个业务过程建立一个事实表，只反映一个业务过程的事实；对于多事务事实表，在同一个事实表中反映多个业务过程。多个业务过程是否放到同一个事实表中，首先需要分析不同业务过程之间的相似性和业务源系统。比如淘宝交易的下单、支付和成功完结这三个业务过程是存在相似性的，都属于订单处理中的一环，并且都来自于交易系统，因此适合放到同一个事务事实表中。

### 2. 粒度和维度

在考虑是采用单事务事实表还是多事务事实表时，另一个关键点就是粒度和维度，在确定好业务过程后，需要基于不同的业务过程确定粒度和维度，当不同业务过程的粒度相同，同时拥有相似的维度时，此时就可以考虑采用多事务事实表。如果粒度不同，则必定是不同的事实表。比如交易中支付和发货有不同的粒度，则无法将发货业务过程放到淘宝交易事务事实表中。

### 3. 事实

对于不同的业务过程，事实往往是不同的，单事务事实表在处理事实上比较方便和灵活，仅仅体现同一个业务过程的事实即可；而多事务事实表由于有多个业务过程，所以有更多的事实需要处理。如果单一业务过程的事实较多，同时不同业务过程的事实又不相同，则可以考虑使用单事务事实表，处理更加清晰；若使用多事务事实表，则会导致事实表零值或空值字段较多。

### 4. 下游业务使用

单事务事实表对于下游用户而言更容易理解，关注哪个业务过程就使用相应的事务事实表；而多事务事实表包含多个业务过程，用户使用

时往往较为困惑。1688 和淘宝交易分别采用了这两种方式，从日常使用来看，对于淘宝交易事务事实表下游用户确实有一定的学习成本。

## 5. 计算存储成本

针对多个业务过程设计事务事实表，是采用单事务事实表还是多事务事实表，对于数据仓库的计算存储成本也是参考点之一，当业务过程数据来源于同一个业务系统，具有相同的粒度和维度，且维度较多而事实相对不多时，此时可以考虑使用多事务事实表，不仅其加工计算成本较低，同时在存储上也相对节省，是一种较优的处理方式。

两种事务事实表的比较如表 11.2 所示。

表 11.2 单事务事实表和多事务事实表的比较

	单事务事实表	多事务事实表
业务过程	一个	多个
粒度	相互间不相关	相同粒度
维度	相互间不相关	一致
事实	只取当前业务过程中的事实	保留多个业务过程中的事实，非当前业务过程中的事实需要置零处理
冗余维度	多个业务过程，则需要冗余多次	不同的业务过程只需要冗余一次
理解程度	易于理解，不会混淆	难以理解，需要通过标签来限定
计算存储成本	较多，每个业务过程都需要计算存储一次	较少，不同业务过程融合到一起，降低了存储计算量，但是非当前业务过程的度量存在大量零值

## 11.2.5 父子事实的处理方式

淘宝交易父子订单的含义在前文确定粒度时有所说明，在同一个店铺同时下单多种商品，不仅每种商品有一个子订单，而且这几个子订单会再单独产生一个父订单。下单和支付都是在父订单粒度上完成的，比如拍下时的订单总额、支付总额、支付邮费，淘宝交易事务事实表在粒度选择上，按照粒度最细原则，确定为子订单，因此需要将下单总额或

者支付总额分摊到每个子订单上,当然只有一个子订单时是不需要进行分摊的。下面以子订单分摊的有效下单金额和支付金额为例加以说明。

子订单下单金额=下单商品数量×商品价格

子订单分摊的有效下单金额=下单商品数量×商品价格+父订单邮费×下  
单分摊比例-子订单折扣-父订单折扣×下单分摊比例

下单分摊比例=(下单商品数量×原价-子订单折扣)/sum(下单商品数量×  
原价-子订单折扣)

子订单分摊的支付金额=父订单支付金额×支付分摊比例

支付分摊比例=(下单商品数量×原价-子订单折扣+调价)/sum(下单商品  
数量×原价-子订单折扣+调价)

通过分摊父订单的金额将所有业务过程的度量全部带进淘宝交易事务事实表中,包括下单数量、商品价格、子订单折扣、下单分摊比例、父订单支付金额、父订单支付邮费、父订单折扣、子订单下单金额、子订单下单有效金额、支付分摊比例、子订单支付金额等,将父子事实同时冗余到事务表中。

## 11.2.6 事实的设计准则

### 1. 事实完整性

事实表包含与其描述的过程有关的所有事实,即尽可能多地获取所有的度量。在淘宝交易事务事实表中,比如支付业务过程,在子订单粒度上的支付金额、支付邮费、支付红包、支付积分、支付折扣都有所包含,覆盖全面。

### 2. 事实一致性

在确定事务事实表的事实时,明确存储每一个事实以确保度量的一致性。以淘宝交易事务事实表为例,在下单业务过程中,有下单商品数量和商品价格两个事实,但在事实表中计算了下单金额和下单有效金

额，它们可以通过商品数量乘以商品价格进行计算。虽然下游在取数时也可以通过这种方式完成计算，但是在事实表中统一计算可以保证度量的一致性，其他如支付过程中的分摊金额等也是类似的。

### 3. 事实可加性

事实表确定事实时，往往会遇到非可加性度量，比如分摊比例、利润率等，虽然它们也是下游分析的关键点，但往往在事务事实表中关注更多的是可加性事实，下游用户在聚合统计时更加方便。在淘宝交易事务事实表中，存储了分摊比例这样的度量，但更多的是存储各类金额的度量。

## 11.3 周期快照事实表

前面章节对事务事实表进行了详细的阐述，同时给出了淘宝交易事务事实表的设计过程。事务事实表可以很好地跟踪一个事件，并对其进行度量，以提供丰富的分析能力。然而，当需要一些状态度量时，比如账户余额、买卖家星级、商品库存、卖家累积交易额等，则需要聚集与之相关的事务才能进行识别计算；或者聚集事务无法识别，比如温度等。对于这些状态度量，事务事实表是无效率的，而这些度量也和度量事务本身一样是有用的，因此，维度建模理论给出了第二种常见的事实表——周期快照事实表，简称“快照事实表”。快照事实表在确定的间隔内对实体的度量进行抽样，这样可以很容易地研究实体的度量值，而不需要聚集长期的事务历史。接下来将以淘宝交易结束后的评价数据、卖家的累积支付金额、买卖家星级等事实表的设计为例，介绍快照事实表在阿里巴巴数据仓库中的设计与应用。

### 11.3.1 特性

快照事实表的设计有一些区别于事务事实表设计的性质。事务事实表的粒度能以多种方式表达，但快照事实表的粒度通常以维度形式声明；事务事实表是稀疏的，但快照事实表是稠密的；事务事实表中的事实是完全可加的，但快照模型将至少包含一个用来展示半可加性质的事实。

#### 1. 用快照采样状态

快照事实表以预定的间隔采样状态度量。这种间隔联合一个或多个维度，将被用来定义快照事实表的粒度，每行都将包含记录所涉及状态的事实。

现在以淘宝交易卖家自然年汇总事实表为例进行介绍。淘宝活动运营小二或者卖家经常都需要看一些交易状态数据，比如自然年至今或者历史至今的下单金额、支付金额、支付买家数、支付商品件数等状态度量，对于卖家而言，可能每天早上都想看一下截至昨天的成交情况；对于小二而言，可能在频繁的活动周期就需要查看一次成交情况。这些状态度量可以每天通过事务事实表进行聚集，但随着时间跨度变大，聚集效率会越来越低，因此需要设计快照事实表进行状态的度量。这里用于采样的周期间隔是每天，如图 11.16 所示的快照事实表记录了每个卖家的下单和支付情况。

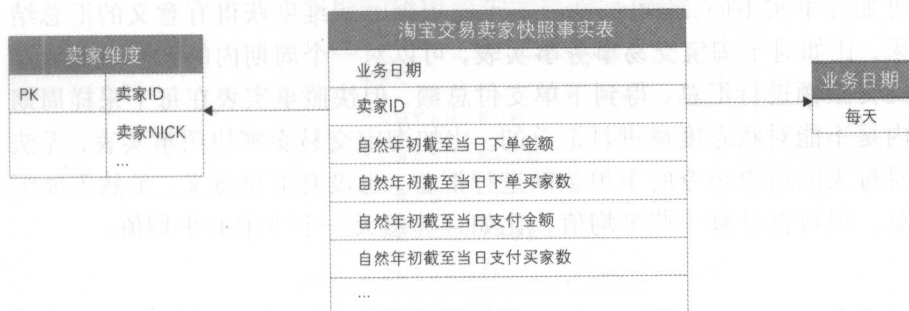


图 11.16 淘宝交易卖家快照事实表



## 2. 快照粒度

事务事实表的粒度可以通过业务过程中所涉及的细节程度来描述，但快照事实表的粒度通常总是被多维声明，可以简单地理解为快照需要采样的周期以及什么将被采样。在淘宝交易卖家快照事实表中，粒度可以被理解为每天针对卖家的历史截至当日的下单支付金额进行快照。

当然，快照周期不一定都按天来进行，也可以按照月或者季度来统计。比如淘宝交易有针对卖家加类目的每月汇总事实表，每月统计一次，同时维度也不仅一个，包含了卖家和类目。

## 3. 密度与稀疏性

快照事实表和事务事实表的一个关键区别在密度上。事务事实表是稀疏的，只有当天发生的业务过程，事实表才会记录该业务过程的事实，如下单、支付等；而快照事实表是稠密的，无论当天是否有业务过程发生，都会记录一行，比如针对卖家的历史至今的下单和支付金额，无论当天卖家是否有下单支付事实，都会给该卖家记录一行。稠密性是快照事实表的重要特征，如果在每个快照周期内不记录行，比如和事务事实表一样，那么确定状态将变得非常困难。

## 4. 半可加性

在快照事实表中收集到的状态度量都是半可加的。与事务事实表的可加性事实不同，半可加性事实不能根据时间维度获得有意义的汇总结果。比如对于淘宝交易事务事实表，可以对一个周期内的下单金额或者支付金额进行汇总，得到下单支付总额，但快照事实表在每个采样周期内是不能对状态度量进行汇总的。比如淘宝交易卖家快照事实表，无法对每天的历史至今的下单金额进行汇总，也没有汇总意义。虽然不能汇总，但可以计算一些平均值，比如计算每天一个下单的平均值。

### 11.3.2 实例

阿里巴巴数据仓库建模时，针对不同的业务场景，事务事实表无法

满足所有的需求。正如上一节所介绍的，在统计历史至今的卖家或者类目的下单金额和支付子订单数时，通过事务事实表聚集效率较低，而采用周期快照事实表则是可行的方案。接下来主要介绍阿里巴巴数据仓库几类周期快照事实表的设计过程。

通过上一节对快照事实表的特性介绍，对于快照事实表的设计步骤可以归纳为：

- 确定快照事实表的快照粒度。
- 确定快照事实表采样的状态度量。

下面将依照该设计步骤介绍几类常见的快照事实表。

1. 单维度的每天快照事实表

(1) 确定粒度

采样周期为每天，针对卖家、买家、商品、类目、地区等维度的快照事实表，比如淘宝卖家历史至今汇总事实表、淘宝商品自然月至今汇总事实表等，不同的采样粒度确定了不同的快照事实表。

(2) 确定状态度量

确定好粒度以后，就要针对这个粒度确定需要采样的状态度量。比如淘宝卖家历史至今汇总事实表，包含了历史截至当日的下单金额、历史截至当日的支付金额等度量，如图 11.17 所示。

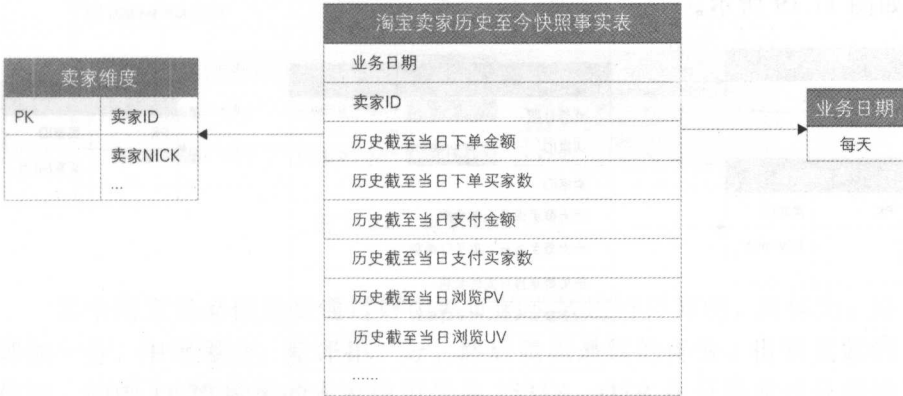


图 11.17 淘宝卖家历史至今快照事实表

淘宝商品历史至今快照事实表，确定了商品维度和商品状态度量，如图 11.18 所示。

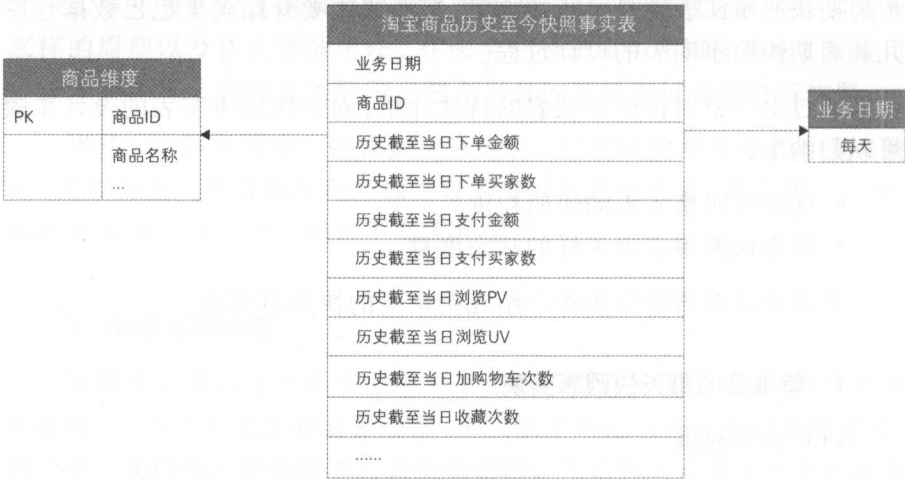


图 11.18 淘宝商品历史至今快照事实表

2. 混合维度的每天快照事实表

混合维度相对于单维度，只是在每天的采样周期上针对多个维度进行采样。比如淘宝买卖家历史至今快照事实表，采样周期依然是每天，维度是卖家加买家，反映的是不同买家对于不同卖家的下单支付金额，如图 11.19 所示。



图 11.19 淘宝买卖家历史至今快照事实表

以上两类快照事实表都有一个特点——都可以从事务事实表进行汇总产出，这是周期快照事实表常见的一种产出模式。除此之外，还有一种产出模式，即直接使用操作型系统的数据作为周期快照事实表的数据源进行加工，比如淘宝卖家星级、卖家 DSR 事实表等。下面介绍这类事实表的设计过程。

### ● 淘宝卖家信用分和 DSR 快照事实表

在介绍这类事实表之前，首先介绍淘宝卖家服务评价系统。在淘宝店铺交易成功以后会进行一次好中差评以及 DSR 评价（包括物流服务、描述相符和服务态度），淘宝卖家信用就是基于好中差评计算得出的，DSR 评分会累积计算一个综合分；天猫店铺交易完成以后仅有 DSR 评价，默认都是好评，如图 11.20、图 11.21 所示。

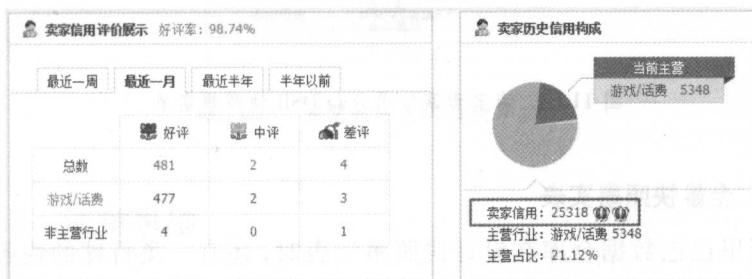


图 11.20 卖家好中差评以及星级

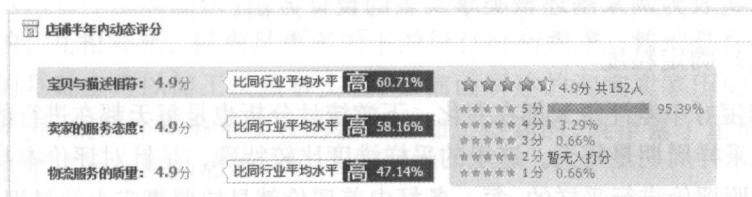


图 11.21 卖家 DSR 评分

其中淘宝卖家信用是通过好中差评的次数进行计算的，具体为：好评加一分，中评零分，差评扣一分；然后累积最终的得分，得到卖家的信用，如图 11.20 所示的卖家信用分是 25318。DSR 评分是通过分项的星级综合得到最终的评分，其中描述相符、服务态度和物流服务都是 1~

5 星的打分方式，综合每一个星级的买家数得到最终的一个平均分，具体为： $(1 \text{ 星} \times 1 \text{ 星人数} + 2 \text{ 星} \times 2 \text{ 星人数} + 3 \text{ 星} \times 3 \text{ 星人数} + 4 \text{ 星} \times 4 \text{ 星人数} + 5 \text{ 星} \times 5 \text{ 星人数}) / (1 \text{ 星人数} + 2 \text{ 星人数} + 3 \text{ 星人数} + 4 \text{ 星人数} + 5 \text{ 星人数})$ 。

前面所述的卖家信用分和 DSR 评分都是在操作型系统中计算完成的，阿里巴巴数据仓库关于淘宝卖家信用分和 DSR 快照事实表是直接采用操作型系统数据进行设计加工，采样周期是每天，针对卖家维度的统计，状态度量就是卖家信用分和 DSR 评分，如图 11.22 所示。



图 11.22 淘宝卖家信用分和 DSR 快照事实表

### 3. 全量快照事实表

阿里巴巴数据仓库在设计快照事实表时，还有一类特殊的快照事实表，即全量快照事实表，这类事实表的特性与前面所述的快照事实表有一些差异，但依然属于周期快照事实表范畴。下面还是以淘宝好中差评快照事实表为例来阐述该类事实表的设计方法。

#### (1) 确定粒度

淘宝好中差评每天都在变化，下游统计分析也是每天都在进行的，因此确定采样周期是每天。这里的采样维度比较特殊，是针对评价本身，即每天按照评价进行采样的，每一条好中差评就是快照事实表的最细粒度。

#### (2) 确定状态度量

对于好中差评评价的度量关注更多的是评价本身，即没有类似于金额、商品数这样的度量，因此设计为无事实的事实表，更多关注评价的状态。

对于全量快照事实表，这里再增加一步，即冗余维度。此如好中差评快照事实表，冗余了子订单维度、商品维度、评论者维度、被评论者

维度以及杂项维度，包括评论内容、是否匿名等信息，如图 11.23 所示。

淘宝好中差评快照事实表	
业务日期	
PK	评论ID
子订单维度	子订单ID 子订单属性
评论者维度	评论者ID 评论者Nick
被评论者维度	被评论者ID 被评论者Nick
商品维度	商品ID 商品名称 商品价格 商品类型
杂项维度	评论内容 是否匿名 业务类型 业务状态

图 11.23 淘宝好中差评快照事实表

11.3.3 注意事项

1. 事务与快照成对设计

数据仓库维度建模时，对于事务事实表和快照事实表往往都是成对设计的，互相补充，以满足更多的下游统计分析需求，特别是在事务事实表的基础上可以加工快照事实表，如前面所述的淘宝卖家历史至今快照事实表，就是在事务事实表的基础上加工得到的，既丰富了星形模型，又降低了下游分析的成本。

2. 附加事实

快照事实表在确定状态度量时，一般都是保存采样周期结束时的状态度量。但是也有分析需求需要关注上一个采样周期结束时的状态度量，而又不愿意多次使用快照事实表，因此一般在设计周期快照事实表时会附加一些上一个采样周期的状态度量。

### 3. 周期到日期度量

在介绍淘宝卖家历史至今快照事实表时，指定了统计周期是卖家历史至今的一些状态度量，比如历史截至当日的下单金额、成交金额等。然而在实际应用中，也有需要关注自然年至今、季度至今、财年至今的一些状态度量，因此在确定周期快照事实表的度量时，也要考虑类似的度量值，以满足更多的统计分析需求。阿里巴巴数据仓库在设计周期快照事实表时，就针对多种周期到日期的度量设计了不同的快照事实表，比如淘宝卖家财年至今的下单金额、淘宝商品自然年至今的收藏次数等。

## 11.4 累积快照事实表

针对淘宝交易，设计了淘宝交易下单/支付/确认收货事务事实表，用于统计下单/支付/确认收货的子订单数、GMV 等。但仍然有很多需求，此事务事实表很难满足，比如统计买家下单到支付的时长、买家支付到卖家发货的时长、买家从下单到确认收货的时长等。如果使用事务事实表进行统计，则逻辑复杂且性能很差。对于类似于研究事件之间时间间隔的需求，采用累积快照事实表可以很好地解决。

### 11.4.1 设计过程

对于累积快照事实表，其建模过程和事务事实表相同，适用于维度建模的步骤。下面详述淘宝交易累积快照事实表的设计过程，并讨论和事务事实表的设计差异。

第一步：选择业务过程。在“事实表基础”一节中讲解了淘宝交易订单的流转过程，主要有四个事件，即买家下单、买家支付、卖家发货、买家确认收货业务过程。对于这四个业务过程，在事务统计中只关注下单、支付和确认收货三个业务过程；而在统计事件时间间隔的需求中，卖家发货也是关键环节。所以针对淘宝交易累积快照事实表，我们选择

这四个业务过程。

第二步：确定粒度。在“事务事实表”中提到，对于淘宝交易，业务需求一般是从子订单粒度进行统计分析，所以选择子订单粒度。淘宝交易事务事实表的粒度也是子订单，但通常对于子订单的每个事件都会记录一行，对于多事件事实表，如果子订单同一周期发生多次事件则记录一行；而对于累积快照事实表，用于考察实体的唯一实例，所以子订单在此表中只有一行记录，事件发生时，对此实例进行更新。

第三步：确定维度。与事务事实表相同，维度主要有买家、卖家、店铺、商品、类目、发货地区、收货地区等。四个业务过程对应的时间字段，格式为日期+时间，分别为下单时间、支付时间、发货时间、确认收货时间，对应于日期维表，图 11.24 中未标识。在实际使用时会使用视图或 SQL 别名的方式表示四个日期角色维度，类似于发货地区维度和收货地区维度。

在交易订单表中，存在很多与订单相关的属性，如订单类型、子类型、支付状态、物流状态、attributes、options 等。对于类似的属性字段，无法归属到已有的商品等维度中，所以新建杂项维度存放。在数据仓库建模理论中，杂项维度无自然键，一般是枚举值的组合，对于每个组合生成一个代理键。但在实际建模中，存在很多非枚举值，且对于每个订单都不相同，如订单的 attributes 和 options 属性。所以实际中杂项维度设计时，也可以使用自然键标识具体的维度值，如图 11.24 中所示的子订单维度和父订单维度。

第四步：确定事实。对于累积快照事实表，需要将各业务过程对应的事实均放入事实表中。比如淘宝交易累积快照事实表，包含了各业务过程对应的事实，如下单对应的下单金额，支付对应的折扣、邮费和支付金额，确认收货对应的金额等。累积快照事实表解决的最重要的问题是统计不同业务过程之间的时间间隔，建议将每个过程的时间间隔作为事实放在事实表中。在淘宝交易累积快照事实表建模中，由于每个过程的时间间隔计算逻辑简单，因此并未加入事实表中，如图 11.25 所示。





图 11.24 淘宝交易累积快照事实表（确定事实前）

第五步：退化维度。在大数据的事实表模型设计中，更多的是考虑提高下游用户的使用效率，降低数据获取的复杂性，减少关联的表数量。一方面，存储成本降低了，而相比之下 CPU 成本仍然较高；另一方面，在大数据时代，很多维表比事实表还大，如淘宝几十亿的商品、几亿的买家等，在分布式数据仓库系统中，事实表和维表关联的成本很高。所以在传统的维度模型设计完成之后，在物理实现中将各维度的常用属性退化到事实表中，以大大提高对事实表的过滤查询、统计聚合等操作的效率，具体详情不再赘述。

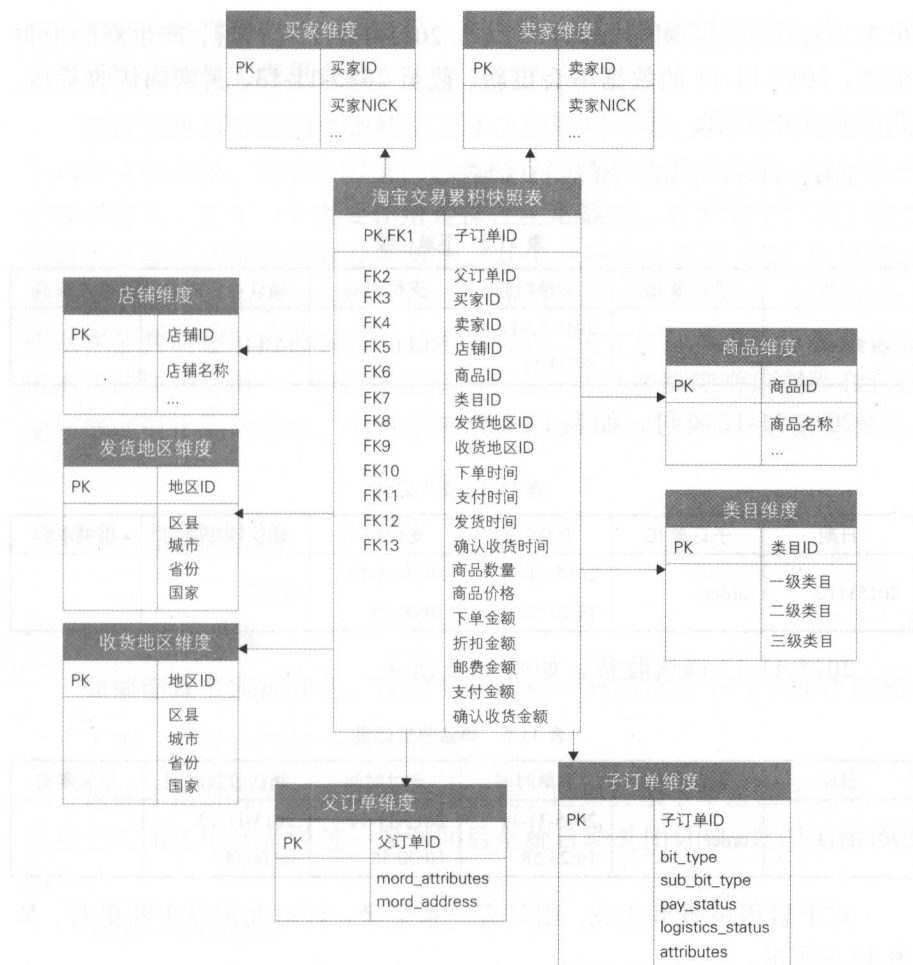


图 11.25 淘宝交易累积快照事实表（确定事实后）

## 11.4.2 特点

### 1. 数据不断更新

事务事实表记录事务发生时的状态，对于实体的某一实例不再更新，而累积快照事实表则对实体的某一实例定期更新。以淘宝交易为例，表 11.3、表 11.4 和表 11.5 通过实例展示了事务事实表的情况，假设采

用多事务事实表：对于 order1 订单，2015-11-12 支付后，产生新的支付记录，2015-11-11 的数据不会更新。截至 2015-11-13，买家确认收货后，共产生 3 条记录。

2015-11-11 下单，如表 11.3 所示。

表 11.3 下单记录

日期	子订单 ID	下单时间	支付时间	确认收货时间	相关事实
20151111	order1	2015-11-11 19:23:58	NULL	NULL	...

2015-11-12 支付，如表 11.4 所示。

表 11.4 支付记录

日期	子订单 ID	下单时间	支付时间	确认收货时间	相关事实
20151112	order1	2015-11-11 19:23:58	2015-11-12 10:00:58	NULL	...

2015-11-13 确认收货，如表 11.5 所示。

表 11.5 确认收货记录

日期	子订单 ID	下单时间	支付时间	确认收货时间	相关事实
20151113	order1	2015-11-11 19:23:58	2015-11-12 10:00:58	2015-11-13 09:00:00	...

对于累积快照事实表，则只有一条记录，针对此记录不断更新，如表 11.6 所示。

表 11.6 累积快照事实表的记录

日期	子订单 ID	下单时间	支付时间	确认收货时间	相关事实
20151113	order1	2015-11-11 19:23:58	2015-11-12 10:00:58	2015-11-13 09:00:00	...

2. 多业务过程日期

累积快照事实表适用于具有较明确起止时间的短生命周期的实体，比如交易订单、物流订单等，对于实体的每一个实例，都会经历从诞生

到消亡等一系列步骤。对于商品、用户等具有长生命周期的实体，一般采用周期快照事实表更合适。

累积快照事实表的典型特征是多业务过程日期，用于计算业务过程之间的时间间隔。但结合阿里巴巴数据仓库模型建设的经验，对于累积快照事实表，还有一个重要作用是保存全量数据。对于淘宝交易，需要保留历史截至当前的所有交易数据，其中一种方式是在 ODS 层保留和源系统结构完全相同的数据；但由于使用时需要关联维度，较为麻烦，所以在公共明细层需要保留一份全量数据，淘宝交易累积快照事实表就承担了这样的作用——存放加工后的事实，并将各维度常用属性和订单杂项维度退化到此表中。通常用于数据探查、统计分析、数据挖掘等。

### 11.4.3 特殊处理

#### 1. 非线性过程

如前面章节所提到的，淘宝交易流程一般经过如下四个业务过程：

下单→支付→发货→确认收货

但并不是所有的交易都会走此流程。比如买家下单之后不支付，可以自己关闭订单或者经过一段时间后系统自动关闭订单。此时交易流程如下：

下单→关闭订单

买家下单并支付之后，可以申请退款，卖家同意后，交易关闭。此时交易流程如下：

下单→支付→关闭订单

在特殊情况下，流程可能会回转。比如在退款过程中，正常流程可能是：

买家申请退款→卖家同意退款→退款达成

或者

买家申请退款→卖家不同意退款→退款关闭

但由于买家和卖家之间未达成协议，卖家不同意买家的退款，此时流程可能是：

买家申请退款→卖家不同意退款→买家申请退款→卖家不同意退款……一直到退款达成或关闭

针对非线性过程，处理情况主要有以下几种。

### (1) 业务过程的统一

比如流程结束标志的统一，最开始设计交易累积快照事实表时，以交易完成作为结束标志；进一步了解业务之后，发现交易关闭也是交易结束的一个分支，所以将交易结束作为流程结束、实体消亡的标志，包括交易完成和交易结束两种情况。

### (2) 针对业务关键里程碑构建全面的流程

比如淘宝交易，全流程可能是下单→支付→发货→确认收货。对于没有支付或没有发货的交易订单，全流程仍然可以覆盖，相关业务过程的时间字段和事实置空。

### (3) 循环流程的处理

主要问题是解决一个业务过程存在多个里程碑日期的问题。使用业务过程第一次发生的日期还是最后一次发生的日期，决定权在商业用户，而不是设计或开发人员。

## 2. 多源过程

针对多业务过程建模时，业务过程可能来自于不同的系统或者来源于不同的表，其对于累积快照事实表的模型设计没有影响，但会影响ETL开发的复杂程度。对于淘宝交易累积快照事实表，除了上述提到的下单→支付→发货→确认收货流程，假设需要关注交易子订单退款业务或者物流业务，此时会涉及交易、售后、物流三个业务源系统。

退款业务流程如下：

下单→支付→买家申请退款→卖家同意退款→退款达成→交易关闭

或者

下单→支付→发货→买家申请退款→卖家同意退款→退款达成→交易关闭

或者

下单→支付→发货→买家申请退款→卖家不同意退款→退款取消→交易成功

针对多源业务建模,主要考虑事实表的粒度问题。对于淘宝交易累积快照事实表,其粒度是交易子订单。对于退款,由于每个子订单可能存在多次退款,此时如果要将退款相关业务过程加入模型中,则需要和商业用户确定存在多次退款时如何取舍,确保模型粒度不变。

### 3. 业务过程取舍

上一节提到的退款业务流程是简化的,比较完整的业务流程如下:

申请退款→申请小二介入→小二实际介入→卖家同意退款→退款完结

将退款相关业务流程设计进入交易累积快照事实表时,是否需要所有的业务过程?答案是否定的。当拥有大量的业务过程时,模型的实现复杂度会增加,特别是对于多源业务过程,模型的耦合度过高,此时需要根据商业用户需求,选取关键的里程碑。

## 11.4.4 物理实现

逻辑模型和物理模型密不可分,针对累积快照事实表模型设计,其有不同的实现方式。

第一种方式是全量表的形式。此全量表一般为日期分区表,每天的分区存储昨天的全量数据和当天的增量数据合并的结果,保证每条记录的状态最新。此种方式适用于全量数据较少的情况。如果数据量很大,此全量表数据量不断膨胀,存储了大量永远不再更新的历史数据,对ETL和分析统计性能影响较大。

第二种方式是全量表的变化形式。此种方式主要针对事实表数据量很大的情况。较短生命周期的业务实体一般从产生到消亡都有一定的时间间隔，可以测算此时间间隔，或者根据商业用户的需求确定一个相对较大的时间间隔。比如针对交易订单，我们以 200 天作为订单从产生到消亡的最大间隔。设计最近 200 天的交易订单累积快照事实表，每天的分区存储最近 200 天的交易订单；而 200 天之前的订单则按照 `gmt_create` 创建分区存储在归档表中。此方式存在的一个问题是 200 天的全量表根据商业需求需要保留多天的分区数据，而由于数据量较大，存储消耗较大。

第三种方式是以业务实体的结束时间分区。每天的分区存放当天结束的数据，设计一个时间非常大的分区，比如 3000-12-31，存放截至当前未结束的数据。由于每天将当天结束的数据归档至当天分区中，时间非常大的分区数据量不会很大，ETL 性能较好；并且无存储的浪费，对于业务实体的某具体实例，在该表的全量数据中唯一。比如对于交易订单，在交易累积快照事实表中唯一。

针对第三种方式，可能存在极特殊情况，即业务系统无法标识业务实体的结束时间。比如业务系统调用接口很多，依赖的系统复杂，最终无法判断业务实体是否已经消亡。如菜鸟的物流订单，由于其依赖物流公司的数据，和大量的物流公司存在接口，按照约定，物流公司会向菜鸟回传运单的流转信息，但无法保证 100% 准确；且一般为批量回传，菜鸟订单系统根据批量数据更新物流订单的结束标志几乎无法实现。前台业务系统没有物流订单的结束时间，那么如何设计物流订单累积快照事实表呢？针对此问题，可以有两种处理方式。

第一种方式，使用相关业务系统的业务实体的结束标志作为此业务系统的结束标志。比如针对物流订单，可以使用交易订单。理论上，交易订单完结了，则物流订单已经完结。

第二种方式，和前端业务系统确定口径或使用前端归档策略。累积快照事实表针对业务实体一般是具有较短生命周期的，和前端业务系统确定口径，确定从业务实体的产生到消亡的最大间隔。另外，针对大量的事实数据，前端系统会定期对历史数据进行归档，避免业务库性能的下降，对于这种情况，可以使用前端系统的归档时间作为业务实体的结束日期。

## 11.5 三种事实表的比较

通过前面章节的介绍，我们对数据仓库三种事实表有了详细的了解。一些业务过程可能只需要一种事实表，而另外一些业务过程可能需要两种或三种事实表。三种事实表相互补充，给出业务的完整描述。表 11.7 对三种事实表进行了比较。

表 11.7 三种事实表的比较

	事务事实表	周期快照事实表	累积快照事实表
时期/时间	离散事务时间点	以有规律的、可预测的间隔产生快照	用于时间跨度不确定的不断变化的工作流
日期维度	事务日期	快照日期	相关业务过程涉及的多个日期
粒度	每行代表实体的一个事务	每行代表某时间周期的一个实体	每行代表一个实体的生命周期
事实	事务事实	累积事实	相关业务过程事实和时间间隔事实
事实表加载	插入	插入	插入与更新
事实表更新	不更新	不更新	业务过程变更时更新

事务事实表记录的事务层面的事实，用于跟踪业务过程的行为，并支持几种描述行为的事实，保存的是最原子的数据，也称为“原子事实表”。事务事实表中的数据在事务事件发生后产生，数据的粒度通常是每个事务一条记录。一旦事务被提交，事实表数据被插入，数据就不能更改，其更新方式为增量更新。

周期快照事实表以具有规律性的、可预见的时间间隔来记录事实，如余额、库存、层级、温度等，时间间隔为每天、每月、每年等，典型的例子如库存日快照表等。周期快照事实表的日期维度通常记录时间段的终止日，记录的事实是这个时间段内一些聚集事实值或状态度量。事实表的数据一旦插入就不能更改，其更新方式为增量更新。

累积快照事实表被用来跟踪实体的一系列业务过程的进展情况，它通常具有多个日期字段，用于研究业务过程中的里程碑过程的时间间隔。另外，它还会有一个用于指示最后更新日期的附加日期字段。由于



事实表中许多日期在首次加载时是不知道的，而且这类事实表在数据加载完成后，可以对其数据进行更新，来补充业务状态变更时的日期信息和事实。

## 11.6 无事实的事实表

在维度模型中，事实表用事实来度量业务过程，不包含事实或度量的事实表称为“无事实的事实表”。虽然没有明确的事实，但可以用来支持业务过程的度量。

常见的无事实的事实表主要有如下两种：

第一种是事件类的，记录事件的发生。在阿里巴巴数据仓库中，最常见的是日志类事实表。比如用户的浏览日志，某会员某时间点浏览了淘宝首页、某会员某时间点浏览了某卖家的店铺中的某商品详情页等。对于每次点击，其事实为 1，但一般不会保存此事实。

第二种是条件、范围或资格类的，记录维度与维度多对多之间的关系。比如客户和销售人员的分配情况、产品的促销范围等。

## 11.7 聚集型事实表

数据仓库的性能是数据仓库建设是否成功的重要标准之一。聚集主要是通过汇总明细粒度数据来获得改进查询性能的效果。通过访问聚集数据，可以减少数据库在响应查询时必须执行的工作量，能够快速响应用户的查询，同时有利于减少不同用户访问明细数据带来的结果不一致问题。尽管聚集能带来良好的收益，但需要事先对其进行加载和维护，这将会对给 ETL 带来更多的挑战。

阿里巴巴将使用频繁的公用数据，通过聚集进行沉淀，比如卖家最

近 1 天的交易汇总表、卖家最近  $N$  天的交易汇总表、卖家自然年交易汇总表等。这类聚集汇总数据，被叫作“公共汇总层”。

在本节中，前半部分将会介绍聚集的基本原则和通用步骤，这些都是在建设聚集型事实表时必须明白的事情；后半部分将会介绍阿里巴巴建设公共汇总层的一些实践。

### 11.7.1 聚集的基本原则

- 一致性。聚集表必须提供与查询明细粒度数据一致的查询结果。从设计角度来看，确保一致性，最简单的方法是确保聚集星形模型中的维度和度量与原始模型中的维度和度量保持一致。
- 避免单一表设计。不要在同一个表中存储不同层次的聚集数据；否则将会导致双重计算或出现更糟糕的事情。在聚集表中有些行存放按天汇总的交易额，有些行存放按月汇总的交易额，这将会让使用者产生误用导致重复计算。为了避免此类问题，通用的做法是在聚集时显式地加入数据层级列以示区别，但是这样会加大使用者的使用成本。行之有效的另一种方法是把按天与按月汇总的交易额用两列存放，但是需要在列名或者列注释上能分辨出来。
- 聚集粒度可不同。聚集并不需要保持与原始明细粒度数据一样的粒度，聚集只关心所需要查询的维度。订单涉及的维度有商品、买家、卖家、地域等，比如可以按照商品汇总一天的交易额，可以按照卖家汇总一天的营业额（交易额），可以按照商品与地域汇总一月的交易额。

### 11.7.2 聚集的基本步骤

#### 第一步：确定聚集维度。

在原始明细模型中会存在多个描述事实的维度，如日期、商品类别、卖家等，这时候需要确定根据什么维度聚集，如果只关心商品的交易额情况，那么就可以根据商品维度聚集数据。

## 第二步：确定一致性上钻。

这时候要关心是按月汇总还是按天汇总，是按照商品汇总还是按照类目汇总，如果按照类目汇总，还需要关心是按照大类汇总还是小类汇总。当然，我们要做的只是了解用户需要什么，然后按照他们想要的进行聚集。

## 第三步：确定聚集事实。

在原始明细模型中可能会有多个事实的度量，比如在交易中有交易额、交易数量等，这时候要明确是按照交易额汇总还是按照成交数量汇总。

### 11.7.3 阿里公共汇总层

#### 1. 基本原则

除了聚集的基本原则外，阿里巴巴建设公共汇总层还必须遵循以下原则。

- 数据公用性。汇总的聚集会有第三者使用吗？基于某个维度的聚集是不是经常用于数据分析中？如果答案是肯定的，那么就有必要把明细数据经过汇总沉淀到聚集表中。
- 不跨数据域。数据域是在较高层次上对数据进行分类聚集的抽象。阿里巴巴以业务过程进行分类，如交易统一到交易域下，商品的新增、修改放到商品域下。
- 区分统计周期。在表的命名上要能说明数据的统计周期，如\_1d表示最近1天，\_td表示截至当天，\_nd表示最近N天。

#### 2. 交易汇总表设计

聚集是指针对原始明细粒度的数据进行汇总。假定已有的交易订单明细模型如图 11.26 所示，可以看出事实和商品、卖家、买家等维度关联。



图 11.26 淘宝交易事务事实表

潜在的聚集如表 11.8 所示。

表 11.8 潜在的聚集

买家维度 (2)	卖家维度 (2)	商品维度 (2)	店铺维度 (2)	类目维度 (3)	发货地区 维度 (5)	.....
买家 ID 买家性别	卖家 ID 卖家性别	商品 ID 商品类型	店铺 ID 店铺类型	类目 ID 一级类目 二级类目	地区 ID 区县 城市 省份 国家	.....

可以看出聚集的组合可能性为各个维度属性个数的乘积： $2 \times 2 \times 2 \times 2 \times 3 \times 5 \dots$ 。下面将按照聚集的基本步骤来介绍聚集表的设计流程。

### (1) 按商品粒度汇总

- 确定聚集维度——商品。
- 确定一致性上钻——按商品（商品 ID）最近 1 天汇总。
- 确定聚集事实——下单量、交易额。

因此，按商品聚集的星形模型如图 11.27 所示。

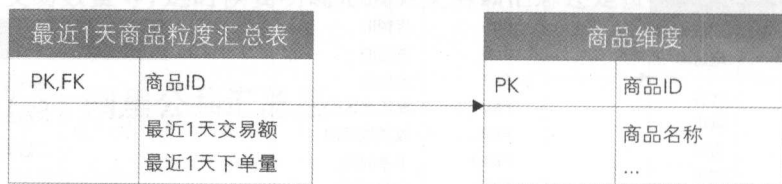


图 11.27 按商品聚集的星形模型

可以看出聚集的事实都是原始模型中的事实，聚集的维度也是原始模型维度中的商品维度，去掉了其他不关心的维度。

### (2) 按卖家粒度汇总

- 确定聚集维度——卖家。
- 确定一致性上钻——按卖家（卖家 ID）最近 7 天和最近 30 天汇总。
- 确定聚集事实——交易额。

因此，按卖家聚集的星形模型如图 11.28 所示。

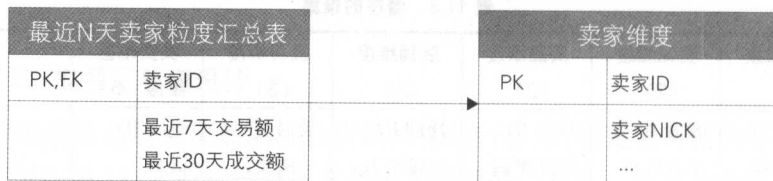


图 11.28 按卖家聚集的星形模型

前面在“聚集的基本原则”中说过，应该避免将不同层级的数据放在一起，为此我们选择用两列存放 7 天和 30 天的事实，但是需要在列名和字段注释上说明清楚。

### (3) 按卖家、买家、商品粒度汇总

- 确定聚集维度——卖家、买家、商品。
- 确定一致性上钻——按卖家（卖家 ID）、买家（买家 ID）、商品（商品 ID）最近 1 天汇总。
- 确定聚集事实——交易额。

因此，按卖家、买家、商品聚集的星形模型如图 11.29 所示。

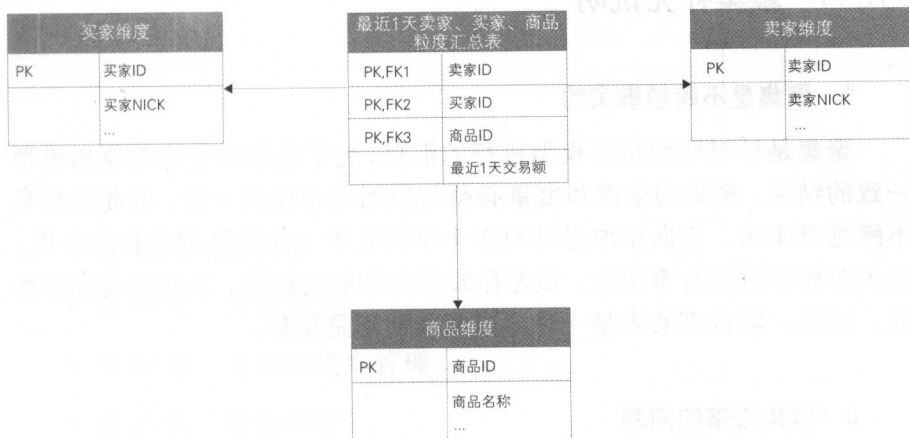


图 11.29 按卖家、买家、商品聚集的星形模型

可以看出聚集的粒度越细，记录的条数越多，就会越接近原始明细模型的粒度。

### (4) 按二级类目汇总

- 确定聚集维度——类目。
- 确定一致性上钻——按最近 1 天类目维度的二级维度属性汇总。
- 确定聚集事实——交易额。

因此，按二级类目聚集的星形模型如图 11.30 所示。

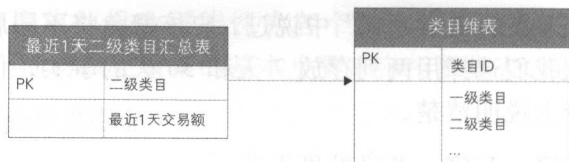


图 11.30 按二级类目聚集的星形模型

与之前的三个聚集表不同的是,这个聚集模型不是根据维度主键属性进行的聚集,而是根据类目的层次维度属性进行的上钻聚集。

## 11.7.4 聚集补充说明

### 1. 聚集是不跨越事实的

聚集是针对原始星形模型进行的汇总,为了获取和查询与原始模型一致的结果,聚集的维度和度量必须与原始模型保持一致,因此聚集是不跨越事实的。横向钻取是针对多个事实基于一致性维度进行的分析,很多时候采用融合事实表,预先存放横向钻取的结果,从而提高查询性能。因此,融合事实表是一种导出模式而不是聚集。

### 2. 聚集带来的问题

聚集会带来查询性能的提升,但聚集也会增加 ETL 维护的难度。当子类目对应的一级类目发生变更时,先前存在的、已经被汇总到聚集表中的数据需要被重新调整。这一额外工作随着业务复杂性的增加,会导致多数 ETL 人员选择简单强力的方法,删除并重新聚集数据。

注:本章节部分理论来自于 Christopher Adamson 的 Star Schema -The Complete Reference 和 Ralph Kimball 的 The Data Warehouse Toolkit-The Definitive Guide to Dimensional Modeling。本书结合阿里的实践进行讲解,细节内容请参考各自著作进行学习。

## 第 3 篇

# 数据管理篇

- 第 12 章 元数据
- 第 13 章 计算管理
- 第 14 章 存储和成本管理
- 第 15 章 数据质量



# 第 12 章

## 元数据

### 12.1 元数据概述

#### 12.1.1 元数据定义

按照传统的定义，元数据（Metadata）是关于数据的数据。元数据打通了源数据、数据仓库、数据应用，记录了数据从产生到消费的全过程。元数据主要记录数据仓库中模型的定义、各层级间的映射关系、监控数据仓库的数据状态及 ETL 的任务运行状态。在数据仓库系统中，元数据可以帮助数据仓库管理员和开发人员非常方便地找到他们所关心的数据，用于指导其进行数据管理和开发工作，提高工作效率。

将元数据按用途的不同分为两类：技术元数据（Technical Metadata）和业务元数据（Business Metadata）。

技术元数据是存储关于数据仓库系统技术细节的数据，是用于开发和管理数据仓库使用的数据。阿里巴巴常见的技术元数据有：

- 分布式计算系统存储元数据,如 MaxCompute 表、列、分区等信息。记录了表的表名。分区信息、责任人信息、文件大小、表类型,生命周期,以及列的字段名、字段类型、字段备注、是否是分区字段等信息。
- 分布式计算系统运行元数据,如 MaxCompute 上所有作业运行等信息;类似于 Hive 的 Job 日志,包括作业类型、实例名称、输入输出、SQL、运行参数、执行时间、最细粒度的 Fuxi Instance (MaxCompute 中 MR 执行的最小单元) 执行信息等。
- 数据开发平台中数据同步、计算任务、任务调度等信息,包括数据同步的输入输出表和字段,以及同步任务本身的节点信息;计算任务主要有输入输出、任务本身的节点信息;任务调度主要有任务的依赖类型、依赖关系等,以及不同类型调度任务的运行日志等。
- 数据质量和运维相关元数据,如任务监控、运维报警、数据质量、故障等信息,包括任务监控运行日志、告警配置及运行日志、故障信息等。

业务元数据从业务角度描述了数据仓库中的数据,它提供了介于使用者和实际系统之间的语义层,使得不懂计算机技术的业务人员也能够“读懂”数据仓库中的数据。阿里巴巴常见的业务元数据有:

- OneData 元数据,如维度及属性、业务过程、指标等的规范化定义,用于更好地管理和使用数据。
- 数据应用元数据,如数据报表、数据产品等的配置和运行元数据。

### 12.1.2 元数据价值

元数据有重要的应用价值,是数据管理、数据内容、数据应用的基础,在数据管理方面为集团数据提供在计算、存储、成本、质量、安全、模型等治理领域上的数据支持。例如在计算上可以利用元数据查找超长运行节点,对这些节点进行专项治理,保障基线产出时间。在数据内容方面为集团数据进行数据域、数据主题、业务属性等的提取和分析提供数据素材。例如可以利用元数据构建知识图谱,给数据打标签,清楚地

知道现在有哪些数据。在数据应用方面打通产品及应用链路，保障产品数据准确、及时产出。例如打通 MaxCompute 和应用数据，明确数据资产等级，更有效地保障产品数据。

### 12.1.3 统一元数据体系建设

元数据的质量直接影响到数据管理的准确性，如何把元数据建设好将起到至关重要的作用。元数据建设的目标是打通数据接入到加工，再到数据消费整个链路，规范元数据体系与模型，提供统一的元数据服务出口，保障元数据产出的稳定性和质量。

统一元数据体系建设思路如图 12.1 所示。

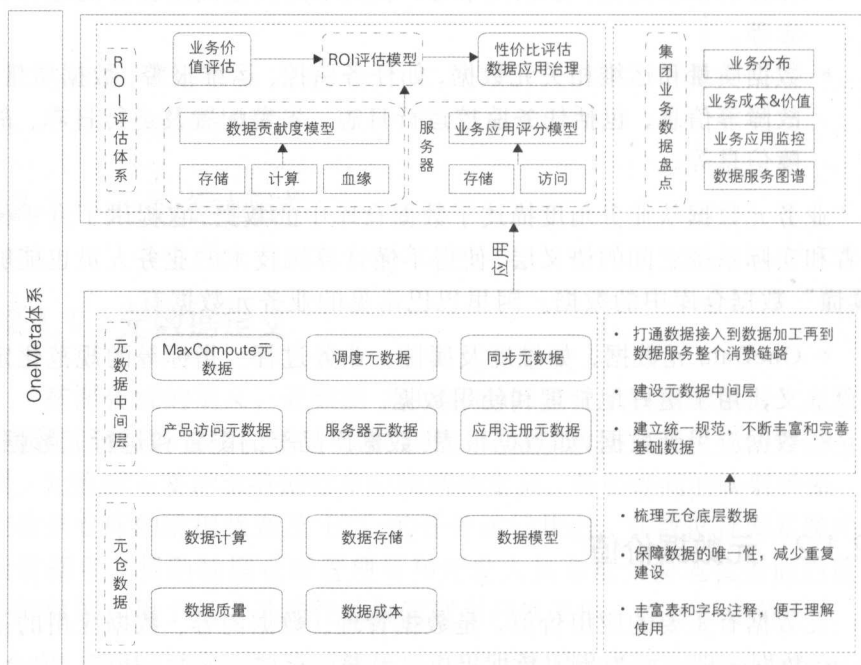


图 12.1 统一元数据体系建设思路图

首先梳理清楚元仓底层数据，对元数据做分类，如计算元数据、存储元数据、质量元数据等，减少数据重复建设，保障数据的唯一性。另

外,要丰富表和字段使用说明,方便使用和理解。根据元仓底层数据构建元仓中间层,依据 OneData 规范,建设元数据基础宽表,也就是元数据中间层,打通从数据产生到消费整个链路,不断丰富中间层数据,如 MaxCompute 元数据、调度元数据、同步元数据、产品访问元数据、服务元数据等。基于元数据中间层,对外提供标准统一的元数据服务出口,保障元数据产出的质量。丰富的元数据中间层不仅能够为集团数据提供在计算、存储、成本、质量、安全、模型等治理领域上的数据支持,形成一套完整的 ROI 数据体系,而且为集团数据进行数据内容、数据域、数据主题、业务属性等的提取和分析提供了数据素材。

## 12.2 元数据应用

数据的真正价值在于数据驱动决策,通过数据指导运营。通过数据驱动的方法,我们能够判断趋势,从而展开有效行动,帮助自己发现问题,推动创新或解决方案的产生。这就是数据化运营。同样,对于元数据,可以用于指导数据相关人员进行日常工作,实现数据化“运营”。比如对于数据使用者,可以通过元数据让其快速找到所需要的数据;对于 ETL 工程师,可以通过元数据指导其进行模型设计、任务优化和任务下线等各种日常 ETL 工作;对于运维工程师,可以通过元数据指导其进行整个集群的存储、计算和系统优化等运维工作。

### 12.2.1 Data Profile

由于阿里巴巴拥有的数据体量实在难以估量,我们很难精确地说清楚到底拥有哪些数据,这些数据存储在哪里,如何使用它们等。过去,数据研发人员在寻找数据、确认口径算法等工序上,花费了大量的人力和时间。

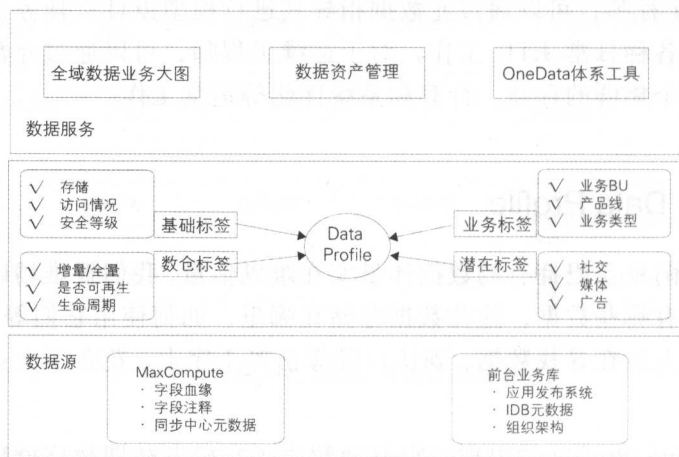
而 Data Profile 的出现,很好地解决了在研发初期数据处理的繁杂困境,既节约了时间成本,同时也缩减了相当一部分人力资源。它的核

心思路是为纷繁复杂的数据建立一个脉络清晰的血缘图谱。通过图计算、标签传播算法等技术，系统化、自动化地对计算与存储平台上的数据进行打标、整理、归档。形象地说，Data Profile 实际承担的是为元数据“画像”的任务。

Data Profile 共有四类标签，就像我们可以为用户的网购行为打上不同的行为标签一样。如果我们也用同样的思维来看待数据本身，那么原本冷冰冰的僵硬数据，实际上也变得有血有肉、个性鲜明。

数据之间的个性化，除了应用场景的不同之外，实际上在数据的研发流程、保障登记、数据质量要求、安全等级、运维策略、告警设置上都会有差异。根据这种差异化，Data Profile 开发出了四类标签，如图 12.2 所示。

- 基础标签：针对数据的存储情况、访问情况、安全等级等进行打标。
- 数仓标签：针对数据是增量还是全量、是否可再生、数据的生命周期来进行标签化处理。
- 业务标签：根据数据归属的主题域、产品线、业务类型为数据打上不同的标签。
- 潜在标签：这类标签主要是为了说明数据潜在的应用场景，比如社交、媒体、广告、电商、金融等。



利用 Data Profile, 不仅可以节约研发人员的时间成本, 同时对阿里巴巴内部的非研发人员来说, 也可以更直观地理解数据、利用数据, 从而提升数据的研发效率。

### 12.2.2 元数据门户

阿里巴巴基于元数据产出的最重要的产品是元数据门户。元数据门户致力打造一站式的数据管理平台、高效的一体化数据市场。包括“前台”和“后台”, “前台”产品为数据地图, 定位消费市场, 实现检索数据、理解数据等“找数据”需求; “后台”产品为数据管理, 定位于一站式数据管理, 实现成本管理、安全管理、质量管理等。

数据地图围绕数据搜索, 服务于数据分析、数据开发、数据挖掘、算法工程师、数据运营等数据表的使用者和拥有者, 提供方便快捷的数据搜索服务, 拥有功能强大的血缘信息及影响分析, 利用表使用说明、评价反馈、表收藏及精品表机制, 为用户浮现高质量、高保障的目标数据。比如在进行数据分析前, 使用数据地图进行关键词搜索, 帮助快速缩小范围, 找到对应的数据; 比如使用数据地图根据表名直接查看表详情, 快速查阅明细信息, 掌握使用规则; 比如通过数据地图的血缘分析可以查看每个数据表的来源、去向, 并查看每个表及字段的加工逻辑。

数据管理平台围绕数据管理, 服务于个人开发者、BU 管理者、系统管理员等用户, 提供个人和 BU 全局资产管理、成本管理和质量管理等。针对个人开发者, 主要包括计算费用和健康分管理、存储费用和健康分管理, 并提供优化建议和优化接口; 针对 BU 管理者和管理员, 主要提供 BU、应用、集群等全局资产消耗概览、分析和预测。

### 12.2.3 应用链路分析

对于某个数据计算任务或表, 其重要程度如何, 是否还有下游在使用, 是否可以下线; 阿里巴巴有这么多种数据产品, 都依赖哪些 MaxCompute 表, 对这些 MaxCompute 表是否需要根据应用的重要程度进行资源、运维保障……对于这些问题, 我们都可以通过元数据血缘来

分析产品及应用的链路,通过血缘链路可以清楚地统计到某个产品所用到的数据在计算、存储、质量上存在哪些问题,通过治理优化保障产品数据的稳定性。

通过应用链路分析,产出表级血缘、字段血缘和表的应用血缘。其中表级血缘主要有两种计算方式:一种是通过 MaxCompute 任务日志进行解析;一种是根据任务依赖进行解析。

其中难度最大的是表的应用血缘解析,其依赖不同的应用。按照应用和物理表的配置关系,可以分为配置型和无配置型。对于数据报表、集市等应用,其数据源直接或间接使用 MaxCompute 数据且有元数据配置依赖关系,通过配置元数据,可以获取 MaxCompute 物理表和具体的报表、集市等应用的血缘关系。对于生意参谋等数据产品,其数据源通过数据同步方式同步至 MySQL、HBase 等数据库,间接使用 MaxCompute 数据且无配置产品和 MySQL、HBase 等物理数据源的依赖关系,导致无法通过配置元数据解析 MaxCompute 数据和数据产品的关系。主要通过统一的应用日志打点 SDK 来解决此问题,可以做到配置化、应用无痕化。

常见的应用链路分析应用主要有影响分析、重要性分析、下线分析、链路分析、寻根溯源、故障排查等。

## 12.2.4 数据建模

传统的数据仓库建模一般采用经验建模的方式,效率较低且不准确。基于现有底层数据已经有下游使用的情况,我们可以通过下游所使用的元数据指导数据参考建模。通过元数据驱动的数据仓库模型建设,可以在一定程度上解决此问题,提高数据仓库建模的数据化指导,提升建模效率。

所使用的元数据主要有:

- 表的基础元数据,包括下游情况、查询次数、关联次数、聚合次数、产出时间等。
- 表的关联关系元数据,包括关联表、关联类型、关联字段、关联次数等。

- 表的字段的基础元数据，包括字段名称、字段注释、查询次数、关联次数、聚合次数、过滤次数等。

其中查询指 SQL 的 SELECT，关联指 SQL 的 JOIN，聚合指 SQL 的 GROUP BY，过滤指 SQL 的 WHERE。

在星形模型设计过程中，可能类似于如下使用元数据。

- 基于下游使用中关联次数大于某个阈值的表或查询次数大于某个阈值的表等元数据信息，筛选用于数据模型建设的表。
- 基于表的字段元数据，如字段中的时间字段、字段在下游使用中的过滤次数等，选择业务过程标识字段。
- 基于主从表的关联关系、关联次数，确定和主表关联的从表。
- 基于主从表的字段使用情况，如字段的查询次数、过滤次数、关联次数、聚合次数等，确定哪些字段进入目标模型。

### 12.2.5 驱动 ETL 开发

通过元数据，指导 ETL 工作，提高 ETL 的效率。在“数据同步”章节中，我们提到了通过元数据驱动一键、批量高效数据同步的 OneClick。OneClick 覆盖的另一个场景是存量数据日常维护，其主要功能如图 12.3 中的“数据运维”部分所示。

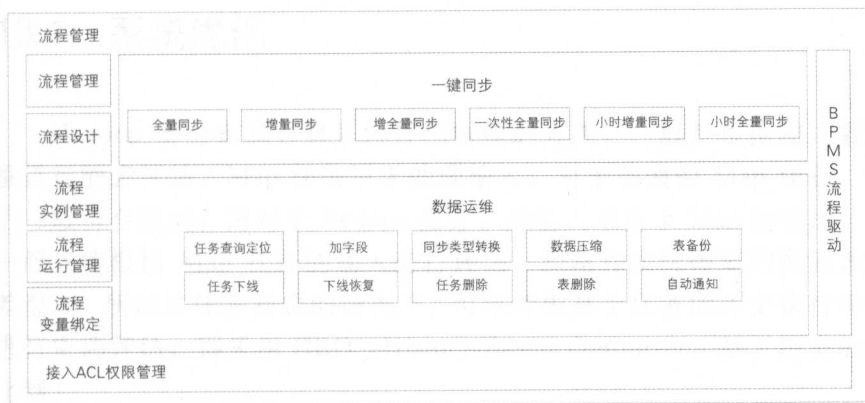


图 12.3 驱动 ETL 开发示意图



我们可以通过 Data Profile 得到数据的下游任务依赖情况、最近被读写的次数、数据是否可再生、每天消耗的存储计算等，这些信息足以让我们判断数据是否可以下线；如果根据一些规则判断可以下线，则会通过 OneClick 触发一个数据下线的工作任务流，数据 Owner 可能只需要点击提交按钮，删除数据、删除元数据、下线调度任务、下线 DQC 监控等一系列操作就会自动在后台执行完成。

# 第 13 章

## 计算管理

目前内部 MaxCompute 集群上有 200 多万个任务，每天存储资源、计算资源消耗都很大。如何降低计算资源的消耗，提高任务执行的性能，提升任务产出的时间，是计算平台和 ETL 开发工程师孜孜追求的目标。本章分别从系统优化和任务优化层面介绍计算优化。

### 13.1 系统优化

Hadoop 等分布式计算系统评估资源的方式，一般是根据输入数据量进行静态评估，Map 任务用于处理输入，对于普通的 Map 任务，评估一般符合预期；而对于 Reduce 任务，其输入来自于 Map 的输出，但一般只能根据 Map 任务的输入进行评估，经常和实际需要的资源数相差很大，所以在任务稳定的情况下，可以考虑基于任务的历史执行情况进行资源评估，即采用 HBO (History-Based Optimizer，基于历史的优化器)。

提到 CBO (Cost-Based Optimizer，基于代价的优化器)，首先会想

到 Oracle 的 CBO。Oracle 会根据收集到的表、分区、索引等统计信息来计算每种执行方式的代价 (Cost)，进而选择其中最优的执行方式。一般来说，对于更多的、更准确的统计信息，CBO 则可能生成代价更小的执行计划。但对表和列上统计信息的收集也是有代价的，尤其是在大数据环境下，表的体量巨大，消耗大量资源收集的统计信息利用率很低。MaxCompute 采用各种抽样统计算法，通过较少的资源获得大量的统计信息，基于先进的优化模型，具备了完善的 CBO 能力，与传统的大数据计算系统相比，性能提升明显。

### 13.1.1 HBO

HBO 是根据任务历史执行情况为任务分配更合理的资源，包括内存、CPU 以及 Instance 个数。HBO 是对集群资源分配的一种优化，概括起来就是：任务执行历史 + 集群状态信息 + 优化规则 → 更优的执行配置。

#### 1. 背景

##### (1) MaxCompute 原资源分配策略

首先看一下 MaxCompute 最初是如何分配 MR 执行过程的 Instance 个数的。默认的 Instance 分配算法如表 13.1 所示。

表 13.1 默认的 Instance 分配算法

Task 类型	最大 Instance 个数	Instance 分配算法
MapTask	99999	$\text{Input\_Table\_Size} / \text{Map\_Split\_Size}$
ReduceTask/JoinTask	1111	主要依据父 Task 的 Instance 的数量进行评估

在这个分配算法的基础上，根据历史数据统计，各个 Instance 处理的数据量分布如下：

##### • Map Instance

```
>fivenum(round(rt$map_input_bytes/1024/1024,2))
```

```
[1] 0.00 4.11 16.59 60.66 4921.94
```

- Reduce Instance

```
>fivenum(round(rt$reduce_input_bytes/1024/1024,2))
[1] 0.00 0.00 0.75 24.87 192721.83
```

- Join Instance

```
>fivenum(round(rt$join_input_bytes/1024/1024,2))
[1] 0.00 0.02 1.82 22.15 101640.31
```

**注解：**fivenum 是 R 语言中统计数据分布的函数，统计值有 5 个，分别是最小值、下四分位数、中位数、上四分位数、最大值。  
\$map\_input\_bytes、\$reduce\_input\_bytes、\$join\_input\_bytes 分别表示 Map、Reduce、Join 三种 Task 的输入数据量 (Bytes)。

从上面内容可以看出，大部分 Instance 处理的数据量远远没有达到预期，即一个 Instance 处理 256MB 的数据。同时有些 Instance 处理的数据量很大，很容易导致任务长尾。

假如把处理数据量小的任务称作小任务，处理数据量大的任务称作大任务，总结：在默认的 Instance 算法下，小任务存在资源浪费，而大任务却资源不足。综上所述，需要有更合理的方法来进行资源分配，HBO 应运而生。

## (2) HBO 的提出

通过数据分析，发现在系统中存在大量的周期性调度的脚本（物理计划稳定），且这些脚本的输入一般比较稳定，如果能对这部分脚本进行优化，那么对整个集群的计算资源的使用率将会得到显著提升。由此，我们想到了 HBO，根据任务的执行历史为其分配更合理的计算资源。HBO 一般通过自适应调整系统参数来达到控制计算资源的目的。

## 2. HBO 原理

HBO 分配资源的步骤如下：

- 前提：最近 7 天内任务代码没有发生变更且任务运行 4 次。
- Instance 分配逻辑：基础资源估算值+加权资源估算值。

### (1) 基础资源数量的逻辑

- 对于 Map Task, 系统需要初始化不同的输入数据量, 根据期望的每个 Map 能处理的数据量, 再结合用户提交任务的输入数据量, 就可以估算出用户提交的任务所需要的 Map 数量。为了保证集群上任务的整体吞吐量, 保证集群的资源不会被一些超大任务占有, 我们采用分层的方式, 提供平均每个 Map 能处理的数据量。
- 对于 Reduce Task, 比较 Hive 使用 Map 输入数据量, MaxCompute 使用最近 7 天 Reduce 对应 Map 的平均输出数据量作为 Reduce 的输入数据量, 用于计算 Instance 的数量。对于 Reduce 个数的估算与 Map 估算基本相同, 不再赘述。

### (2) 加权资源数量的逻辑

- 对于 Map Task, 系统需要初始化期望的每个 Map 能处理的数据量。通过该 Map 在最近一段时间内的平均处理速度与系统设定的期望值做比较, 如果平均处理速度小于期望值, 则按照同等比例对基础资源数量进行加权, 估算出该 Map 的加权资源数量。
- 对于 Reduce Task, 方法同上。

最终的 Instance 个数为: 基础资源估算值+加权资源估算值。

- CPU/内存分配逻辑: 类似于 Instance 分配逻辑, 也是采用基础资源估算值+加权资源估算值的方法。

## 3. HBO 效果

### (1) 提高 CPU 利用率

通过适当降低每个 Instance 的 CPU 资源数, 集群利用率从 40% 提升到 80%。其中早上 4:00—7:00 节省的 CPU 资源可以供 6 万个 Instance 并发使用。

### (2) 提高内存利用率

在保障并行度, 同时又能提高执行效率的基础上, 合理分配内存, 早上 4:00—7:00 节省的内存资源可以供 4 万个 Instance 并发使用。

### (3) 提高 Instance 并发数

合理设置 Task 的 Instance 个数, Instance 峰值并发数提升了 57%。

### (4) 降低执行时长

在某机器上测试效果很明显。该集群有 3700 台机器, 任务数约 16 万个, 总执行时长减少 4472 小时 (没有开启 HBO 时总执行时长是 8356 小时, 开启 HBO 后总执行时长为 3884 小时)。

## 4. HBO 改进与优化

HBO 是基于执行历史来设置计划的, 对于日常来说, 数据量波动不大, 工作良好。但是某些任务在特定场合下依旧有数据量暴涨的情况发生, 尤其是在大促“双 11”和“双 12”期间, 这个日常生成的 HBO 计划就不适用了。针对这个问题, HBO 也增加了根据数据量动态调整 Instance 数的功能, 主要依据 Map 的数据量增长情况进行调整。

## 13.1.2 CBO

MaxCompute 2.0 引入了基于代价的优化器 (CBO), 根据收集的统计信息来计算每种执行方式的代价, 进而选择最优的执行方式。该优化器对性能提升做出了卓越改进。通过性能评测, MaxCompute 2.0 离线计算比同类产品 Hive 2.0 on Tez 快 90%以上。

### 1. 优化器原理

优化器 (Optimizer) 引入了 Volcano 模型 (请参考论文: *The Volcano Optimizer Generator: Extensibility and Efficient Search*), 该模型是基于代价的优化器 (CBO), 并且引入了重新排序 Join (Join Reorder) 和自动 MapJoin (Auto MapJoin) 优化规则等, 同时基于 Volcano 模型的优化器会尽最大的搜索宽度来获取最优计划。

优化器有多个模块相互组合协调工作, 包括 Meta Manager (元数据)、Statistics (统计信息)、Rule Set (优化规则集)、Volcano Planner Core

(核心计划器)等,如图 13.1 所示。

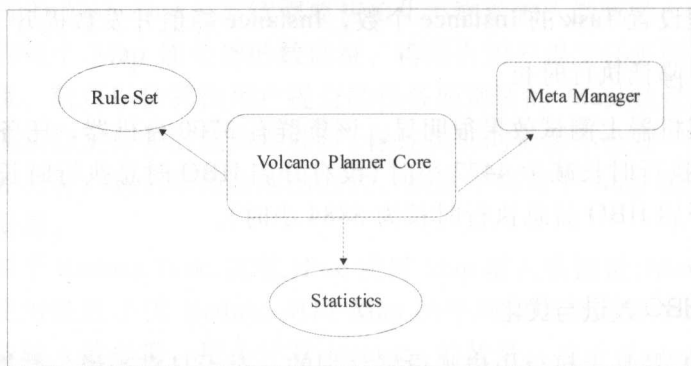


图 13.1 优化器各模块协调工作图

### (1) Meta Manager

Meta 模块主要提供元数据信息,包括表的元数据、统计信息元数据等。当优化器在选择计划时,需要根据元数据的一些信息进行优化。比如表分区裁剪 (TableScan Partition Pruning) 优化时,需要通过 Meta 信息获取表数据有哪些分区,然后根据过滤条件来裁剪分区。同时还有一些基本的元数据,如表是否是分区表、表有哪些列等。

对于 Meta 的管理,MaxCompute 提供了 Meta Manager 与优化器进行交互。Meta Manager 与底层的 Meta 部分对接,提供了优化器所需要的信息。

### (2) Statistics

Statistics 主要是帮助优化器选择计划时,提供准确的统计信息,如表的 Count 值、列的 Distinct 值、TopN 值等。优化器只有拥有准确的统计信息,才能计算出真正最优的计划。比如 Join 是选择 Hash Join 还是 Merge Join,优化器会根据 Join 的输入数据量 (即 Count 值) 来进行选择。

优化器提供了 UDF 来收集统计信息,包括 Distinct 值、TopN 值等,而 Count 值等统计信息是由底层 Meta 直接提供的。

### (3) Rule Set

优化规则是根据不同情况选择不同的优化点,然后由优化器根据代价模型 (Cost Model) 来选择启用哪些优化规则。比如工程合并规则 (Project Merge Rule), 将临近的两个 Project 合并成一个 Project; 过滤条件下推规则 (Filter Push Down), 将过滤条件尽量下推, 使得数据先进行过滤, 然后再进行其他计算, 以减少其他操作的数据量。这些所有的优化都放置在优化规则集中。

MaxCompute 优化器提供了大量的优化规则, 用户也可以通过 set 等命令来控制所使用的规则。规则被分为 Substitute Rule (被认为是优化了肯定好的规则)、Explore Rule (优化后需要考虑多种优化结果)、Build Rule (可以认为优化后的结果不能再次使用规则进行优化)。

### (4) Volcano Planner Core

Volcano Planner 是整个优化器的灵魂, 它会将所有信息 (Meta 信息、统计信息、规则) 统一起来处理, 然后根据代价模型的计算, 获得一个最优计划。

① 代价模型。代价模型会根据不同操作符 (如 Join、Project 等) 计算不同的代价, 然后计算出整个计划中最小代价的计划。MaxCompute 代价模型目前提供的 Cost 由三个维度组成, 即行数、I/O 开销、CPU 开销, 通过这三个维度来衡量每一个操作符的代价。

② 工作原理。假设 Planner 的输入是一棵由 Compiler 解析好的计划树, 简称 RelNode 树, 每个节点简称 RelNode。

#### • Volcano Planner 创建

Planner 的创建主要是将 Planner 在优化过程中要用到的信息传递给执行计划器, 比如规则集, 用户指定要使用的规划; Meta Provider, 每个 RelNode 的 Meta 计算, 如 RowCount 值计算、Distinct 值计算等; 代价模型, 计算每个 RelNode 的代价等。这些都是为以后 Planner 提供的必要信息。

#### • Planner 优化

**规则匹配 (Rule Match):** 是指 RelNode 满足规则的优化条件而建



立的一种匹配关系。Planner 首先将整个 RelNode 树的每一个 RelNode 注册到 Planner 内部，同时在注册过程中，会在规则集中找到与每个 RelNode 匹配的规则，然后加入到规则应用 (Rule Apply) 的队列中。所以整个注册过程处理结束后，所有与 RelNode 可以匹配的规则全部加入到队列中，以后应用时只要从队列中取出来就可以了。

**规则应用 (Rule Apply):** 是指从规则队列 (Rule Queue) 中弹出 (Pop) 一个已经匹配成功的规则进行优化。当获取到一个已经匹配的规则进行处理时，如果规则优化成功，则肯定会产生至少一个新的 RelNode，因为进行了优化，所以与之前未优化时的 RelNode 有差异。这时需要再次进行注册以及规则匹配操作，把与新产生的 RelNode 匹配的规则加入到规则队列中，然后接着下次规则应用。

Planner 会一直应用所有的规则，包括后来叠加的规则，直到不会有新的规则匹配到。至此，整个优化结束，这时就可以找到一个最优计划。

**代价计算 (Cost Compute):** 每当规则应用之后，如果规则优化成功，则会产生新的 RelNode。在新的 RelNode 注册过程中，有一个步骤是计算 RelNode 的代价。

代价计算由代价模型对每个 RelNode 的代价进行估算。

- 如果不存在代价，或者 Child 的代价还没有估算（默认是最大值），则忽略。
- 如果存在代价，则会将本身的代价和 Child（即输入的所有 RelNode）的代价进行累加，若小于 Best，则认为优化后的 RelNode 是当前最优的。并且会对其 Parent 进行递归估算代价，即传播代价计算 (Propagate Calculate Cost)。

比如计划：Project->TableScan，当 TableScan 计算代价为 1 时，则会继续估算 Project 的代价，假设为 1，则整个 Project 的代价就是  $1+1=2$ 。

也就是说，当 RelNode 本身的代价估算出来后，会递归地对 Parent 进行代价估算，这样就可以对整条链路的计划进行估算。在这个估算过程中借助了 Meta Manager 和 Statistics 提供的信息（见图 13.2）。

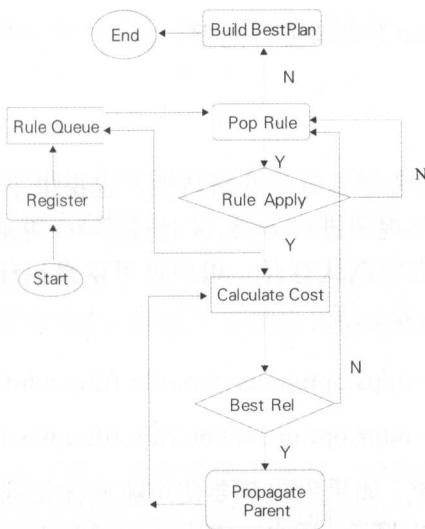


图 13.2 Volcano Planner 工作原理图

## 2. 优化器新特性

优化器具有一些新特性，主要是重新排序 Join (Join Reorder) 和自动 MapJoin (Auto MapJoin)。

### (1) 重新排序 Join

Join 可以认为是关系数据库中最重要操作符之一。Join 的性能也直接关系到整个 SQL 的性能。

重新排序 Join 可以认为是将 Join 的所有不同输入进行一个全排列，然后找到代价最小的一个排列。之前仅仅保持了用户书写 SQL 语句的 Join 顺序，这样的 Join 顺序不一定是最优的，所以通过重新排序 Join 规则可以实现更好的选择，提供更优的性能。

### (2) 自动 MapJoin

Join 实现算法有多种，目前主要有 Merge Join 和 MapJoin。对于小数据量，MapJoin 比 Merge Join 性能更优。之前是通过 Hint 方式来指定是否使用 MapJoin，这样对用户不是很友好，且使用不方便。自动 MapJoin 充分利用优化器的代价模型进行估算，获得更优的 MapJoin 方

式，而不是通过 Hint 方式来进行处理。

### 3. 优化器使用

MaxCompute 优化器具有一些新特性，也提供了许多优化规则，将内部已经实现的优化规则进行分类，并且提供 set 等命令方便用户使用。一些基础优化规则都会默认打开，用户也可以自己任意搭配使用。

优化器提供的 Flag 有：

规则白名单——odps.optimizer.cbo.rule.filter.white

规则黑名单——odps.optimizer.cbo.rule.filter.black

使用方法很简单，如果用户需要使用哪些优化规则，只要将规则的缩写名称加入白名单即可；反之，需要关闭哪些优化规则，只要将名称加入黑名单即可。比如 set odps.optimizer.cbo.rule.filter.black=xxx,yyy;，就表示将 xxx,yyy 对应的优化规则关闭。

对于重新排序 Join 和自动 MapJoin，对应的标记分别是 porj 和 hj。即如果想使用上述优化，则可以进行如下设置：

```
set odps.optimizer.cbo.rule.filter.white=porj,hj;
```

注：优化规则之间请使用“,”分隔。

下面列举 TPC-H 的一些测试结果，如图 13.3 所示。

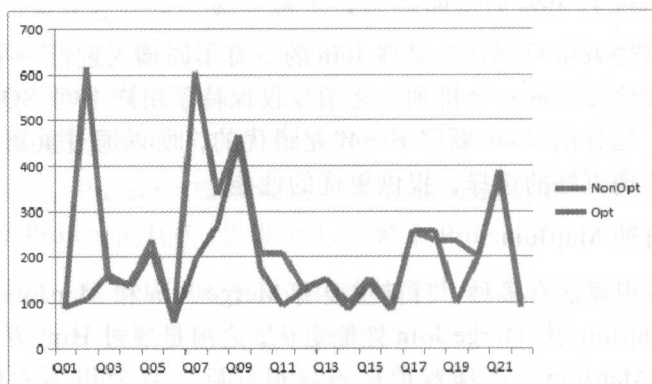


图 13.3 TPC-H 测试对比图

优化前：5129s，优化后：3814s，性能提升：25.7%。

#### 4. 注意事项

由于用户书写 SQL 语句时可能存在一些不确定因素，所以应尽量避免这些因素带来的性能影响，甚至结果非预期。

Optimizer 会提供谓词下推 (Predicate Push Down) 优化，主要目的是尽量早地进行谓词过滤，以减少后续操作的数据量，提高性能。但需要注意的是：

##### (1) UDF

对于 UDF 是否下推，优化器做了限制，不会任意下推这种带有用户意图的函数，主要是因为不同用户书写的函数含义不一样，不可以一概而论。如果用户需要下推 UDF，则要自己改动 SQL，这样做主要的好处是用户自己控制 UDF 执行的逻辑，最了解自己的 UDF 使用在 SQL 的哪个部分，而不是优化器任意下推等。例如 UDF 可能和数据顺序有关，下推和不下推会导致出现不同的结果。理论上，过滤条件可以下推到 Exchange 之后，但不是所有 UDF 都是这样的；否则会导致结果违背了用户书写 SQL 语句的本意。

##### (2) 不确定函数

对于不确定函数，优化器也不会任意下推，比如 sample 函数，如果用户将其写在 where 子句中，同时语句存在 Join，则优化器是不会下推到 TableScan 的，因为这样可能改变了原意。

例如：

```
SELECT *  
FROM t1  
JOIN t2  
ON t1.c1=t2.d1  
WHERE sample(4, 1) = true;
```

则 sample 函数在 Join 之后执行，而不会直接在 TableScan 后执行。

如果用户需要对 TableScan 进行抽样,则需要自己修改 SQL 来达到目的;否则优化器进行下推可能会错误地理解用户的意图。

对上述 SQL 语句修改如下:

```
SELECT *
FROM
(
    SELECT *
    FROM t1
    WHERE sample(4, 1) = true
) t1
JOIN t2
ON t1.c1=t2.d1;
```

### (3) 隐式类型转换

书写 SQL 语句时,应尽量避免 Join Key 存在隐式类型转换。例如 String = Bigint, 在这种情况下会转换为 ToDouble(String)=ToDouble(Bigint), 这是不是用户原本的意图,数据库本身不得而知。这样可能引发的后果有两种:一种是转换失败,报错;另一种是虽然执行成功了,但结果与用户期望的不一致。

## 13.2 任务优化

本节主要从数据倾斜方面讨论数据优化。下面给出一个 SQL/MR 从提交到最后执行在 MaxCompute 中的细分步骤,如图 13.4 所示。

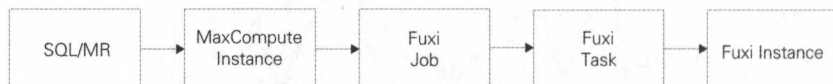


图 13.4 SQL/MR 在 MaxCompute 中的细分步骤

SQL/MR 作业一般会生成 MapReduce 任务，在 MaxCompute 中则会生成 MaxCompute Instance，通过唯一 ID 进行标识。

- Fuxi Job: 对于 MaxCompute Instance，则会生成一个或多个由 Fuxi Task 组成的有向无环图，即 Fuxi Job。MaxCompute Instance 和 Fuxi Job 类似于 Hive 中 Job 的概念。
- Fuxi Task: 主要包含三种类型，分别是 Map、Reduce 和 Join，类似于 Hive 中 Task 的概念。
- Fuxi Instance: 真正的计算单元，和 Hive 中的概念类似，一般和槽位 (slot) 对应。

## 13.2.1 Map 倾斜

### 1. 背景

Map 端是 MR 任务的起始阶段，Map 端的主要功能是从磁盘中将数据读入内存，Map 端的两个主要过程如图 13.5 所示。

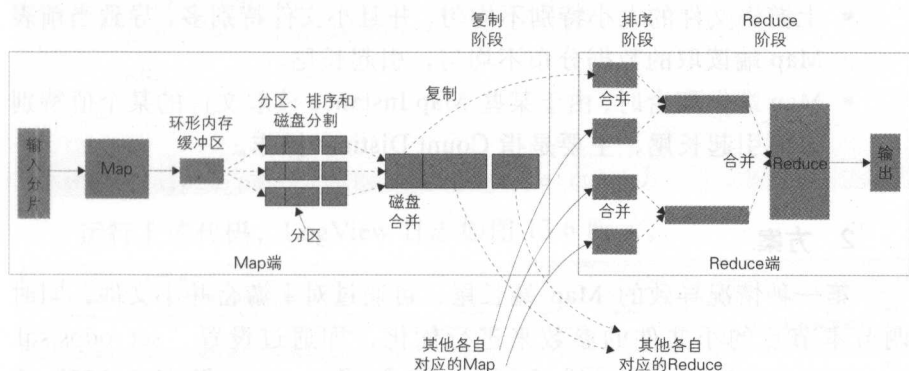


图 13.5 Map 端的两个主要过程示意图

- 每个输入分片会让一个 Map Instance 来处理，在默认情况下，以 Pangu 文件系统的一个文件块的大小（默认为 256MB）为一个分片。Map Instance 输出的结果会暂时放在一个环形内存缓冲区中，当该缓冲区快要溢出时会在本地文件系统中创建一个溢出文件，

即 Write Dump。在 Map 读数据阶段，可以通过“set odps.mapper.split.size=256”来调节 Map Instance 的个数，提高数据读入的效率，同时也可以通过“set odps.mapper.merge.limit.size=64”来控制 Map Instance 读取文件的个数。如果输入数据的文件大小差异比较大，那么每个 Map Instance 读取的数据量和读取时间差异也会很大。

- 在写入磁盘之前，线程首先根据 Reduce Instance 的个数划分分区，数据将会根据 Key 值 Hash 到不同的分区上，一个 Reduce Instance 对应一个分区的数据。Map 端也会做部分聚合操作，以减少输入 Reduce 端的数据量。由于数据是根据 Hash 分配的，因此也会导致有些 Reduce Instance 会分配到大量数据，而有些 Reduce Instance 却分配到很少数据，甚至没有分配到数据。

在 Map 端读数据时，由于读入数据的文件大小分布不均匀，因此会导致有些 Map Instance 读取并且处理的数据特别多，而有些 Map Instance 处理的数据特别少，造成 Map 端长尾。以下两种情况可能会导致 Map 端长尾：

- 上游表文件的大小特别不均匀，并且小文件特别多，导致当前表 Map 端读取的数据分布不均匀，引起长尾。
- Map 端做聚合时，由于某些 Map Instance 读取文件的某个值特别多而引起长尾，主要是指 Count Distinct 操作。

## 2. 方案

第一种情况导致的 Map 端长尾，可通过对上游合并小文件，同时调节本节点的小文件的参数来进行优化，即通过设置“set odps.sql.mapper.merge.limit.size=64”和“set odps.sql.mapper.split.size=256”两个参数来调节，其中第一个参数用于调节 Map 任务的 Map Instance 的个数；第二个参数用于调节单个 Map Instance 读取的小文件个数，防止由于小文件过多导致 Map Instance 读取的数据量很不均匀；两个参数配合调整。下面主要讨论第二种情况的处理方式。

如下代码的作用是获取手机 APP 日志明细中的前一个页面的页面

组信息，其中 `pre_page` 是前一个页面的页面标识，`page_ut` 表是存储的手机 APP 的页面组，`pre_page` 只能通过匹配正则或者所属的页面组信息，进行笛卡儿积 Join。

原始代码：

```
SELECT ...
FROM
(
    SELECT ds
           ,unique_id
           ,pre_page
    FROM tmp_app_ut_1
    WHERE ds='${bizdate}'
    AND pre_page is not null
) a
LEFT OUTER JOIN
(
    SELECT t.*
           ,length(t.page_type_rule) rule_length
    FROM page_ut t
    WHERE ds='${bizdate}'
    AND is_enable = 'Y'
) b
ON 1=1
WHERE a.pre_page rlike b.page_type_rule ;
```

运行上述代码，LogView 日志如图 13.6 所示。

The screenshot shows the LogView interface with two main sections. The top section displays job execution details for 'M3\_Stg1' and 'M3\_Stg2'. The bottom section shows a latency chart for 'M3\_Stg1'.

TaskName	Fatal/Finished/TotalInstCount	I/O Records	FinishedPercentage	Status	StartTime	EndTime	Latency(s)	TimeLine
1. L1_Stg4	0/71	448/447	100%	Terminated	2015-10-12 14:09:14	2015-10-12 14:09:22	8	
1. M3_Stg1	0/50	197612626/0		Running	2015-10-12 14:09:42		41:17	
2. R5_3_Stg2	0/51	0/0		Running	2015-10-12 14:09:42		41:17	

FluxInstanceID	IP & Path	StdOut	StdErr	Status	FinishedPercentage	StartTime	EndTime	Latency(s)	TimeLine
1. Ocpu/Running	10.182.143.11...			Running		2015-10-12 14:09:47		41:14	
2. Ocpu/Running	10.182.111.18...			Running		2015-10-12 14:09:47		41:14	
3. Ocpu/Running	10.182.133.12...			Running		2015-10-12 14:09:47		41:14	
4. Ocpu/Running	10.182.105.76...			Running		2015-10-12 14:09:47		41:14	
5. Ocpu/Running	10.182.105.21...			Running		2015-10-12 14:09:47		41:14	
6. Ocpu/Running	10.182.143.11...			Running		2015-10-12 14:09:47		41:14	

图 13.6 LogView 日志 (一)



L1\_Stg4 是 MapJoin 小表的分发阶段；M3\_Stg1 是读取明细日志表的 Map 阶段，与 MapJoin 小表的 Join 操作也发生在这个阶段；R5\_3\_Stg2 是进行分组排序的阶段。

通过日志发现，M3\_Stg1 阶段单个 Instance 的处理时间达到了 40 分钟，而且长尾的 Instance 个数比较固定，应是不同的 Map 读入的文件块分布不均匀导致的，文件块大的 Map 数据量比较大，在与小表进行笛卡儿积操作时，非常耗时，造成 Map 端长尾。针对这种情况，可以使用“distribute by rand()”来打乱数据分布，使数据尽可能分布均匀。

修改后代码如下：

```
SELECT ...
FROM
(
    SELECT ds
           ,unique_id
           ,pre_page
    FROM tmp_app_ut_1
    WHERE ds='${bizdate}'
    AND pre_page is not null
    DISTRIBUTE BY rand()
) a
LEFT OUTER JOIN
(
    SELECT t.*
           ,length(t.page_type_rule) rule_length
    FROM page_ut t
    WHERE ds='${bizdate}'
    AND is_enable = 'Y'
) b
ON 1=1
WHERE a.pre_page rlike b.page_type_rule ;
```

执行上述代码，LogView 日志如图 13.7 所示。

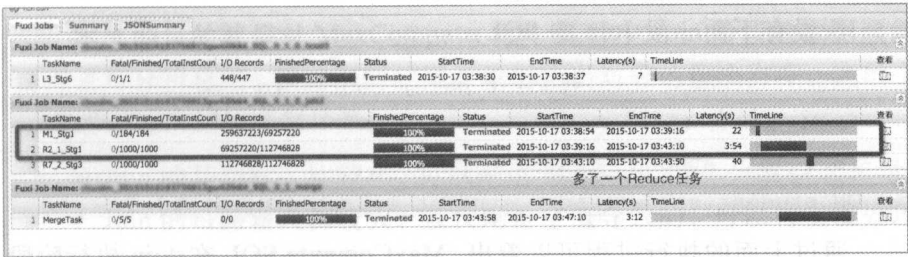


图 13.7 LogView 日志 (二)

通过“`distribute by rand()`”会将 Map 端分发后的数据重新按照随机值再进行一次分发。原先不加随机分发函数时，Map 阶段需要与使用 MapJoin 的小表进行笛卡儿积操作，Map 端完成了大小表的分发和笛卡儿积操作。使用随机分布函数后，Map 端只负责数据的分发，不再有复杂的聚合或者笛卡儿积操作，因此不会导致 Map 端长尾。

3. 思考

Map 端长尾的根本原因是由于读入的文件块的数据分布不均匀，再加上 UDF 函数性能、Join、聚合操作等，导致读入数据量大的 Map Instance 耗时较长。在开发过程中如果遇到 Map 端长尾的情况，首先考虑如何让 Map Instance 读取的数据量足够均匀，然后判断是哪些操作导致 Map Instance 比较慢，最后考虑这些操作是否必须在 Map 端完成，在其他阶段是否会做得更好。

13.2.2 Join 倾斜

1. 背景

Join 操作需要参与 Map 和 Reduce 的整个阶段。首先通过一段 Join 的 SQL 来看整个 Map 和 Reduce 阶段的执行过程以及数据的变化，进而对 Join 的执行原理有所了解。

假设有下面一段 Join 的 SQL:

```
SELECT student_id, student_name, course_id
FROM student
LEFT JOIN student_course
ON student.student_id = student_course.student_id;
```

通过上面的执行过程可以看出, MaxCompute SQL 在 Join 执行阶段会将 Join Key 相同的数据分发到同一个执行 Instance 上处理。如果某个 Key 上的数据量比较大, 则会导致该 Instance 执行时间较长。其表现为: 在执行日志中该 Join Task 的大部分 Instance 都已执行完成, 但少数几个 Instance 一直处于执行中 (这种现象称之为长尾)。



图 13.8 Map 和 Reduce 阶段的执行过程

因为数据倾斜导致长尾的现象比较普遍, 严重影响任务的执行时间, 尤其是在“双 11”等大型活动期间, 长尾程度比平时更严重。比如某些大型店铺的 PV 远远超过一般店铺的 PV, 当用浏览日志数据和卖家维表关联时, 会按照卖家 ID 进行分发, 导致某些大卖家所在 Instance 处理的数据量远远超过其他 Instance, 而整个任务会因为这个长尾的 Instance 迟迟无法结束。

本文的目的就是对 MaxCompute SQL 执行中 Join 阶段的数据倾斜情况进行分析总结, 根据不同的倾斜原因给出对应的解决方案。这里主要讲述三种常见的倾斜场景。

- Join 的某路输入比较小, 可以采用 MapJoin, 避免分发引起长尾。
- Join 的每路输入都较大, 且长尾是空值导致的, 可以将空值处理成随机值, 避免聚集。
- Join 的每路输入都较大, 且长尾是热点值导致的, 可以对热点值和非热点值分别进行处理, 再合并数据。

下面会针对这三种场景给出具体的解决方案。首先我们了解一下如何确认 Join 是否发生数据倾斜。

打开 MaxCompute SQL 执行时产生的 LogView 日志, 点开日志会看到每个 Fuxi Task 的详细执行信息, 如图 13.9 所示。

TaskName	Data/Finished/TotalInstCnt	I/O Records	FinishedPercentage	Status	StartTime	EndTime	Latency(s)	TimeLine
1 M39_Stg1	0/1674/1674	4930702	100%	Terminate	2016-04-21 03:2..	2016-04-21 03:5..	26:32	
2 M2_Stg3	0/3/3	4952008	100%	Terminate	2016-04-21 03:2..	2016-04-21 03:2..	24	
3 M1_Stg3	0/42/42	7957958	100%	Terminate	2016-04-21 03:2..	2016-04-21 03:3..	2:9	
4 M26_Stg11	0/9/9	5500630	100%	Terminate	2016-04-21 03:2..	2016-04-21 03:3..	1:23	
5 M38_Stg12	0/96/96	7328747	100%	Terminate	2016-04-21 03:2..	2016-04-21 03:3..	1:56	
6 M7_Stg5	0/9/9	6870146	100%	Terminate	2016-04-21 03:2..	2016-04-21 03:3..	57	
7 J1_1_2_Stg3	0/1999/1999	7958937	100%	Terminate	2016-04-21 03:3..	2016-04-21 03:3..	33	
8 J10_3_7_St5	0/1999/1999	8645334	100%	Terminate	2016-04-21 03:3..	2016-04-21 03:4..	9:7	
9 J28_10_24..	0/1999/1999	8508021	100%	Terminate	2016-04-21 03:4..	2016-04-21 03:4..	47	

SubTaskName	Data/Finished/TotalInstCnt	I/O Records	FinishedPercentage	Status	StartTime	EndTime	Latency(s)	TimeLine
0 J10_3_7_St..	10.182.82.1..		100%	Terminate	2016-04-21 03:3..	2016-04-21 03:3..	1:16	
1 J10_3_7_St..	10.182.91.9..		100%	Terminate	2016-04-21 03:3..	2016-04-21 03:3..	1:32	
2 J10_3_7_St..	10.182.82.9..		100%	Terminate	2016-04-21 03:3..	2016-04-21 03:3..	34	

图 13.9 Fuxi Task 的详细执行信息

从图 13.9 中可以看到每一个 Map、Join、Reduce 的 Fuxi Task 任务, 点击其中一个 Join 任务, 可以看到有 115 个 Instance 长尾; 再点击 StdOut, 可以查看 Instance 读入的数据量, 如图 13.10 所示。

Read from 0 num: 52743413size: 1389941257

图 13.10 Instance 读入的数据量

图 13.10 中显示 Join 的一路输入读取的数据量是 1389941257 行。如果 Long-Tails 中 Instance 读入的数据量远超过其他 Instance 读取的数据量, 则表示某个 Instance 处理的数据量超大导致长尾。

## 2. 方案

针对上面提到的三种倾斜场景，给出以下三种对应的解决方案。

### (1) MapJoin 方案

Join 倾斜时，如果某路输入比较小，则可以采用 MapJoin 避免倾斜。MapJoin 的原理是将 Join 操作提前到 Map 端执行，将小表读入内存，顺序扫描大表完成 Join。这样可以避免因为分发 key 不均匀导致数据倾斜。但是 MapJoin 的使用有限制，必须是 Join 中的从表比较小才可用。所谓从表，即左外连接中的右表，或者右外连接中的左表。

MapJoin 的使用方法非常简单，在代码中 select 后加上 “/\*+mapjoin(a) \*/” 即可，其中 a 代表小表（或者子查询）的别名。现在 MaxCompute 已经可以自动选择是否使用 MapJoin，可以不使用显式 Hint。例如：

```
SELECT /*+MAPJOIN(b) */
      a.c2
      ,b.c3
FROM
(
    SELECT c1
          ,c2
    FROM   t1
) a
LEFT OUTER JOIN
(
    SELECT c1
          ,c3
    FROM   t2
) b
on      a.c1 = b.c1;
```

另外，使用 MapJoin 时对小表的大小有限制，默认小表读入内存后的大小不能超过 512MB，但是用户可以通过设置 “set odps.sql.mapjoin.memory.max=2048” 加大内存，最大为 2048MB。

## (2) Join 因为空值导致长尾

另外,数据表中经常出现空值的数据,如果关联 key 为空值且数据量比较大,Join 时就会因为空值的聚集导致长尾,针对这种情况可以将空值处理成随机值。因为空值无法关联上,只是分发到一处,因此处理成随机值既不会影响关联结果,也能很好地避免聚焦导致长尾。例如:

```
SELECT ...
FROM          table_a
LEFT OUTER JOIN table_b
ON  coalesce(table_a.key,rand()*9999)=table_b.key  --当
key 为空值时用随机值替代
```

## (3) Join 因为热点值导致长尾

如果是因为热点值导致的长尾,并且 Join 的输入比较大无法使用 MapJoin,则可以先将热点 key 取出,对于主表数据用热点 key 切分成热点数据和非热点数据两部分分别处理,最后合并。这里以淘宝的 PV 日志表关联商品维表取商品属性为例进行介绍。

- 取热点 key: 将 PV 大于 50000 的商品 ID 取出到临时表中。

```
INSERT OVERWRITE TABLE topk_item
SELECT item_id
FROM
(
    SELECT item_id
           ,count(1) as cnt
    FROM   pv      --pv 表
    WHERE  ds = '${bizdate}'
    AND    url_type = 'ipv'
    AND    item_id is not null
    GROUP BY item_id
) a
WHERE    cnt >= 50000
```

- 取出非热点数据。

将主表 (pv 表) 和热点 key 表 (topk\_item 表) 外关联后,通过条件 “bl.item\_id is null” 取关联不到的数据即非热点商品的日志数据,此



时需要使用 MapJoin。再用非热点数据关联商品维表，因为已经排除了热点数据，所以不会存在长尾。

```

SELECT...
FROM
(
    SELECT      *
    FROM        item                --商品表
    WHERE       ds = '${bizdate}'
) a
RIGHT OUTER JOIN
(
    SELECT      /*+MAPJOIN(b1)*/
               b2.*
    FROM
    (
        SELECT  item_id
        FROM    topk_item    --热点表
        WHERE   ds = '${bizdate}'
    ) b1
    RIGHT OUTER JOIN
    (
        SELECT  *
        FROM    pv                --pv表
        WHERE   ds = '${bizdate}'
        AND     url_type = 'ipv'
    ) b2
    ON b1.item_id = coalesce(b2.item_id,concat("tbcdm",
rand()))
    WHERE       b1.item_id is null
) l
ON a.item_id = coalesce(l.item_id,concat("tbcdm",rand()))
    
```

- 取出热点数据。

将主表（pv 表）和热点 key 表（topk\_item 表）内关联，此时需要使用 MapJoin，取到热点商品的日志数据。同时，需要将商品维表（item

表) 和热点 key 表 (topk\_item 表) 内关联, 取到热点商品的维表数据。然后将第一部分数据外关联第二部分数据, 因为第二部分数据只有热点商品的维表, 数据量比较小, 可以使用 MapJoin 避免长尾。

```

SELECT /*+MAPJOIN(a)*/
...
FROM
(
    SELECT /*+MAPJOIN(b1)*/
        b2.*
    FROM
    (
        SELECT item_id
        FROM topk_item
        WHERE ds = '${bizdate}'
    ) b1
    JOIN
    (
        SELECT *
        FROM pv --pv 表
        WHERE ds = '${bizdate}'
        AND url_type = 'ipv'
        AND item_id is not null
    ) b2
    ON (b1.item_id = b2.item_id)
) 1
LEFT OUTER JOIN
(
    SELECT /*+MAPJOIN(a1)*/
        a2.*
    FROM
    (
        SELECT item_id
        FROM topk_item
        WHERE ds = '${bizdate}'
    ) a1
    JOIN
    (

```



```

SELECT *
FROM item      --商品表
WHERE ds = '${bizdate}'

) a2
ON (a1.item_id = a2.item_id)

) a
ON a.item_id = l.item_id
    
```

- 将上面取到的非热点数据和热点数据通过“union all”合并后即得到完整的日志数据，并且关联了商品信息。

针对倾斜问题，MaxCompute 系统也提供了专门的参数用来解决长尾问题，如下所示。

- 开启功能：

```
set odps.sql.skewjoin=true/false
```

- 设置倾斜的 key 及对应的值：

```
set odps.sql.skewinfo=skewed_src: (skewed_key)
[("skewed_value")]
```

其中 skewed\_key 代表倾斜的列，skewed\_value 代表倾斜列上的倾斜值。

设置参数的好处很明显，简单方便；坏处是如果倾斜值发生变化需要修改代码，而且一般无法提前知道变化。另外，如果倾斜值比较多也不方便在参数中设置。需要根据实际情况选择拆分代码或者设置参数。

### 3. 思考

当大表和大表 Join 因为热点值发生倾斜时，虽然可以通过修改代码来解决，但是修改起来很麻烦，代码改动也很大，且影响阅读。而 MaxCompute 现有的参数设置使用不够灵活，倾斜值多的时候不可能将所有值都列在参数中，且倾斜值可能经常变动。因此，我们还一直在探索和思考，期望有更好的、更智能的解决方案，如生成统计信息，MaxCompute 内部根据统计信息来自动生成解决倾斜的代码，避免投入过多的人力。

## 13.2.3 Reduce 倾斜

### 1. 背景

Reduce 端负责的是对 Map 端梳理后的有序 key-value 键值对进行聚合, 即进行 Count、Sum、Avg 等聚合操作, 得到最终聚合的结果。

Distinct 是 MaxCompute SQL 中支持的语法, 用于对字段去重。比如计算在某个时间段内支付买家数、访问 UV 等, 都是需要用 Distinct 进行去重的。MaxCompute 中 Distinct 的执行原理是将需要去重的字段以及 Group By 字段联合作为 key 将数据分发到 Reduce 端。

因为 Distinct 操作, 数据无法在 Map 端的 Shuffle 阶段根据 Group By 先做一次聚合操作, 以减少传输的数据量, 而是将所有的数据都传输到 Reduce 端, 当 key 的数据分发不均匀时, 就会导致 Reduce 端长尾。

Reduce 端产生长尾的主要原因就是 key 的数据分布不均匀。比如有些 Reduce 任务 Instance 处理的数据记录多, 有些处理的数据记录少, 造成 Reduce 端长尾。如下几种情况会造成 Reduce 端长尾:

- 对同一个表按照维度对不同的列进行 Count Distinct 操作, 造成 Map 端数据膨胀, 从而使得下游的 Join 和 Reduce 出现链路上的长尾。
- Map 端直接做聚合时出现 key 值分布不均匀, 造成 Reduce 端长尾。
- 动态分区数过多时可能造成小文件过多, 从而引起 Reduce 端长尾。
- 多个 Distinct 同时出现在一段 SQL 代码中时, 数据会被分发多次, 不仅会造成数据膨胀  $N$  倍, 还会把长尾现象放大  $N$  倍。

### 2. 方案

对于上面提到的第二种情况, 可以对热点 key 进行单独处理, 然后通过“Union All”合并。这种解决方案已经在“Join 倾斜”一节中介绍过。

对于上面提到的第三种情况, 可以把符合不同条件的数据放到不同的分区, 避免通过多次“Insert Overwrite”写入表中, 特别是分区数比较多时, 能够很好地简化代码。但是动态分区也有可能会带来小文件过

多的困扰。以最简 SQL 为例：

```
INSERT OVERWRITE TABLE part_test PARTITION(ds)
SELECT *
FROM part_test;
```

假设有  $K$  个 Map Instance,  $N$  个目标分区：

		ds=1
cfile1		ds=2
...	×	ds=3
cfilek		...
		ds=n

那么在最坏的情况下，可能产生  $K \times N$  个小文件，而过度的小文件会对文件系统造成巨大的管理压力，因此 MaxCompute 对动态分区的处理是引入额外一级的 Reduce Task，把相同的目标分区交由同一个（或少量几个）Reduce Instance 来写入，避免小文件过多，并且这个 Reduce 肯定是最后一个 Reduce Task 操作。MaxCompute 是默认开启这个功能的，也就是将下面参数设置为 true。

```
set odps.sql.reshuffle.dynamiccpt=true;
```

设置这个参数引入额外一级的 Reduce Task 的初衷是为了解决小文件过多的问题，那么如果目标分区数比较少，根本就不会造成小文件过多，这时候默认开启这个功能不仅浪费了计算资源，而且还降低了性能。因此，在此种情况下关闭这个功能：

```
set odps.sql.reshuffle.dynamiccpt=false
```

上面几种情况相对比较简单，这里重点介绍第四种情况。

如图 13.11 所示这段代码是在 7 天、30 天等时间范围内，分 PC 端、无线端、所有终端，计算支付买家数和支付商品数，其中支付买家数和支付商品数指标需要去重。因为需要根据日期、终端等多种条件组合对买家和商品进行去重计算，因此有 12 个 Count Distinct 计算。在计算过程中会根据 12 个组合 key 分发数据来统计支付买家数和支付商品数。这样做使得节点运行效率变低。

```

,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') THEN buyer_id ELSE NULL END) AS pay_ord_byr_cnt_1w_001 --最近 7 天支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') THEN t1.item_id ELSE NULL END) AS pay_ord_byr_cnt_1m_001 --最近 7 天 PC 端支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') AND is_wireless = 'N' THEN buyer_id ELSE NULL END) AS pay_ord_byr_cnt_1m_002 --最近 7 天无线端支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') AND is_wireless = 'Y' THEN buyer_id ELSE NULL END) AS pay_ord_byr_cnt_1m_003 --最近 7 天无线端支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -29, 'dd'), 'yyyymmdd') THEN buyer_id ELSE NULL END) AS pay_ord_byr_cnt_1m_004 --最近 30 天支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -29, 'dd'), 'yyyymmdd') THEN t1.item_id ELSE NULL END) AS pay_ord_byr_cnt_1m_005 --最近 30 天 PC 端支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -29, 'dd'), 'yyyymmdd') AND is_wireless = 'N' THEN buyer_id ELSE NULL END) AS pay_ord_byr_cnt_1m_006 --最近 30 天无线端支付买家数
,COUNT(DISTINCT CASE WHEN t1.ds = TO_CHAR(ATEADDTO_DATE('s'(b1:sdte)), 'yyyymmdd'), -29, 'dd'), 'yyyymmdd') AND is_wireless = 'Y' THEN buyer_id ELSE NULL END) AS pay_ord_byr_cnt_1m_007 --最近 30 天无线端支付买家数

```

图 13.11 在多种时间范围内分终端统计买家数和商品数代码

如图 13.12 所示是该代码的运行 LogView 日志，节点运行时长为 1h14min，数据膨胀。

针对上面的问题，可以先分别进行查询，执行 Group By 原表粒度 + buyer\_id，计算出 PC 端、无线端、所有终端以及在 7 天、30 天等统计口径下的 buyer\_id（这里可以理解为买家支付的次数），然后在子查询外 Group By 原表粒度，当上一步的 Count 值大于 0 时，说明这一买家在这个统计口径下有过支付，计入支付买家数，否则不计入。计算支付商品数采用同样的处理方式。最后对支付商品数和支付买家数进行 Join 操作。

M2_Stg1	0 / 75	1768566100/1768566100
M9_Stg4	0 / 2	12185738/7647679
M1_Stg1	0 / 19826	3556968737/1759617640
J3_1_2_Stg1	0 / 1111	3528183740/18633242208
R5_3_Stg2	0 / 1111	18633242208/12308850

图 13.12 LogView 日志

按上述方案修改后的代码如下（仅示例支付买家数的计算）：

```

SELECT t2.seller_id
      ,t2.price_seg_id
      ,SUM(case when pay_ord_byr_cnt_1w_001>0 then 1 else 0 end)
AS pay_ord_byr_cnt_1w_001 --最近 7 天支付买家数
      ,SUM(case when pay_ord_byr_cnt_1w_002>0 then 1 else 0 end)
AS pay_ord_byr_cnt_1w_002 --最近 7 天 PC 端支付买家数
      ,SUM(case when pay_ord_byr_cnt_1w_003>0 then 1 else 0 end)
AS pay_ord_byr_cnt_1w_003 --最近 7 天无线端支付买家数
      ,SUM(case when pay_ord_byr_cnt_1m_002>0 then 1 else 0 end)

```

```

AS pay_ord_byr_cnt_1m_002 --最近 30 天支付买家数
    ,SUM(case when pay_ord_byr_cnt_1m_003>0 then 1 else 0 end)
AS pay_ord_byr_cnt_1m_003 --最近 30 天 PC 端支付买家数
    ,SUM(case when pay_ord_byr_cnt_1m_004>0 then 1 else 0 end)
AS pay_ord_byr_cnt_1m_004 --最近 30 天无线端支付买家数
FROM
(
    SELECT  a1.seller_id
            ,a2.price_seg_id
            ,buyer_id
            ,COUNT(buyer_id) AS pay_ord_byr_cnt_1m_002 --最近
30 天支付买家数
            ,COUNT(CASE WHEN is_wireless = 'N' THEN buyer_id
ELSE NULL END) AS pay_ord_byr_cnt_1m_003 --最近 30 天 PC 端支付买
家数
            ,COUNT(CASE WHEN is_wireless = 'Y' THEN buyer_id
ELSE NULL END) AS pay_ord_byr_cnt_1m_004 --最近 30 天无线端支付买
家数
            ,COUNT(case
                        when a1.ds>=TO_CHAR(DATEADD(TO_DATE
('$bizdate}', 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') then buyer_id
                        else null
                        end) AS pay_ord_byr_cnt_1w_001 --最近 7 天支
付买家数
            ,COUNT(CASE WHEN a1.ds>=TO_CHAR(DATEADD(TO_DATE
('$bizdate}', 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') and
is_wireless = 'N' THEN buyer_id
                        ELSE NULL
                        END) AS pay_ord_byr_cnt_1w_002 --最近 7 天 PC
端支付买家数
            ,COUNT(CASE WHEN a1.ds>=TO_CHAR(DATEADD(TO_DATE
('$bizdate}', 'yyyymmdd'), -6, 'dd'), 'yyyymmdd') and

```

```

is_wireless = 'Y' THEN buyer_id
                ELSE NULL
            END) AS pay_ord_byr_cnt_1w_003 --最近7天无线端支付买家数
FROM
(
    select *
    from table_pay --支付表
) a1
JOIN ( SELECT item_id
        ,price_seg_id
    FROM tag_itm --商品 tag 表
    WHERE ds = '${bizdate}')
    a2
ON ( a1.item_id = a2.item_id )
GROUP BY a1.seller_id --原表粒度
        ,a2.price_seg_id --原表粒度
        ,buyer_id
)t2
GROUP BY t2.seller_id --原表粒度
        ,t2.price_seg_id; --原表粒度

```

经测试,修改后的运行时间为13min,优化后的效果还是非常明显的。整体运行的LogView日志如图13.13所示,可以看到和Count Distinct计算方式相比数据没有膨胀,约为原方式的1/10。

M10_Stg4	0 /2	12185738/7647679
M2_Stg1	0 /75	1768566100/3537132200
M1_Stg1	0 /19826	3556968737/3519235280
J14_1_2_Stg7	0 /1111	3528183740/1525908931
J3_1_2_Stg1	0 /1111	3528183740/91935869
R5_3_Stg2	0 /1111	91935869/88120790

图 13.13 整体运行的 LogView 日志



### 3. 思考

对 Multi Distinct 的思考如下：

- 上述方案中如果出现多个需要去重的指标，那么在把不同指标 Join 在一起之前，一定要确保指标的粒度是原始表的数据粒度。比如支付买家数和支付商品数，在子查询中指标粒度分别是：原始表的数据粒度 + buyer\_id 和原始表的数据粒度 + item\_id，这时两个指标不是同一数据粒度，所以不能 Join，需要再套一层代码，分别把指标 Group By 到“原始表的数据粒度”，然后再进行 Join 操作。
- 修改前的 Multi Distinct 代码的可读性比较强，代码简洁，便于维护；修改后的代码较为复杂。当出现的 Distinct 个数不多、表的数据量也不是很大、表的数据分布较均匀时，不使用 Multi Distinct 的计算效果也是可以接受的。所以，在性能和代码简洁、可维护之间需要根据具体情况进行权衡。另外，这种代码改动还是比较大的，需要投入一定的时间成本，因此可以考虑做成自动化，通过检测代码、优化代码自动生成将会更加方便。
- 当代码比较臃肿时，也可以将上述子查询落到中间表里，这样数据模型更合理、复用性更强、层次更清晰。当需要去除类似的多个 Distinct 时，也可以查一下是否有更细粒度的表可用，避免重复计算。

目前 Reduce 端数据倾斜很多是由 Count Distinct 问题引起的，因此在 ETL 开发工作中应该予以重视 Count Distinct 问题，避免数据膨胀。对于一些表的 Join 阶段的 Null 值问题，应该对表的数据分布要有清楚的认识，在开发时解决这个问题。

## 第14章

# 存储和成本管理

在大数据时代，移动互联、社交网络、数据分析、云服务等应用迅速普及，对数据中心提出了革命性的需求，存储管理已经成为 IT 核心之一。对于数据爆炸式的增长，存储管理也将面临着一系列挑战。如何有效地降低存储资源的消耗，节省存储成本，将是存储管理孜孜追求的目标。本章主要从数据压缩、数据重分布、存储治理项优化、生命周期管理等的角度介绍存储管理优化。

### 14.1 数据压缩

在分布式文件系统中，为了提高数据的可用性与性能，通常会将数据存储 3 份，这就意味着存储 1TB 的逻辑数据，实际上会占用 3TB 的物理空间。目前 MaxCompute 中提供了 archive 压缩方法，它采用了具有更高压缩比的压缩算法，可以将数据保存为 RAID file 的形式，数据不再简单地保存为 3 份，而是使用盘古 RAID file 的默认值 (6,3) 格式的文件，即 6 份数据+3 份校验块的方式，这样能够有效地将存储比约



为 1:3 提高到 1:1.5, 大约能够省下一半的物理空间。当然, 使用 archive 压缩方式也有一定的风险, 如果某个数据块出现了损坏或者某台机器宕机损坏了, 恢复数据块的时间将要比原来的方式更长, 读的性能会有一些的损失。因此, 目前一般将 archive 压缩方法应用在冷备数据与日志数据的压缩存储上。例如, 一些非常大的淘系日志数据, 底层数据超过一定的时间期限后使用的频率非常低, 但是又是属于不可恢复的重要数据, 对于这部分数据就可以考虑对历史数据的分区进行 archive 压缩, 使用 RAID file 来存储, 以此来节省存储空间。示例如下:

```
alter table A partition(ds='20130101') archive;
```

其输出信息如表 14.1 所示。

表 14.1 输出信息

	File count	File size	File physical size
Before merge	1	456	1366
After merge	1	512	765

在输出信息中可以看到 archive 前后的逻辑存储 (File size) 和物理存储 (File physical size) 的变化情况, 而且在这个过程中会将多个小文件自动合并。

## 14.2 数据重分布

在 MaxCompute 中主要采用基于列存储的方式, 由于每个表的数据分布不同, 插入数据的顺序不一样, 会导致压缩效果有很大的差异, 因此通过修改表的数据重分布, 避免列热点, 将会节省一定的存储空间。目前我们主要通过修改 distribute by 和 sort by 字段的方法进行数据重分布, 如图 14.1 所示。

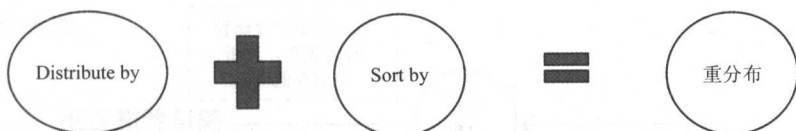


图 14.1 数据重分布方法

数据重分布效果的波动比较大，这主要跟数据表中字段的重复值、字段本身的大小、其他字段的具体分布有一定的关系，一般我们会筛选出重分布效果高于 15% 的表进行优化处理。

重分布前后一些底层大表的效果对比如表 14.2 所示。

表 14.2 重分布前后效果对比

表名	重分布前	重分布后	节约存储	节约比例
Table_A	17.6TB	12.52TB	5.07TB	28.8%
Table_B	10.7TB	4.4TB	6.3TB	59.03%
Table_C	11.54TB	8.645TB	2.898TB	23.1%

## 14.3 存储治理项优化

阿里巴巴数据仓库在资源管理的过程中，经过不断地实践，慢慢摸索出一套适合大数据的存储优化方法，在元数据的基础上，诊断、加工成多个存储治理优化项。目前已有的存储治理优化项有未管理表、空表、最近 62 天未访问表、数据无更新无任务表、数据无更新有任务表、开发库数据大于 100GB 且无访问表、长周期表等。通过该优化项的数据诊断，形成治理项，治理项通过流程的方式进行运转、管理，最终推动各个 ETL 开发人员进行操作，优化存储管理，并及时回收优化的存储效果。在这个体系下，形成现状分析、问题诊断、管理优化、效果反馈的存储治理项优化的闭环。通过这个闭环，可以有效地推进数据存储的优化，降低存储管理的成本。

存储治理项优化的主要流程如图 14.2 所示。

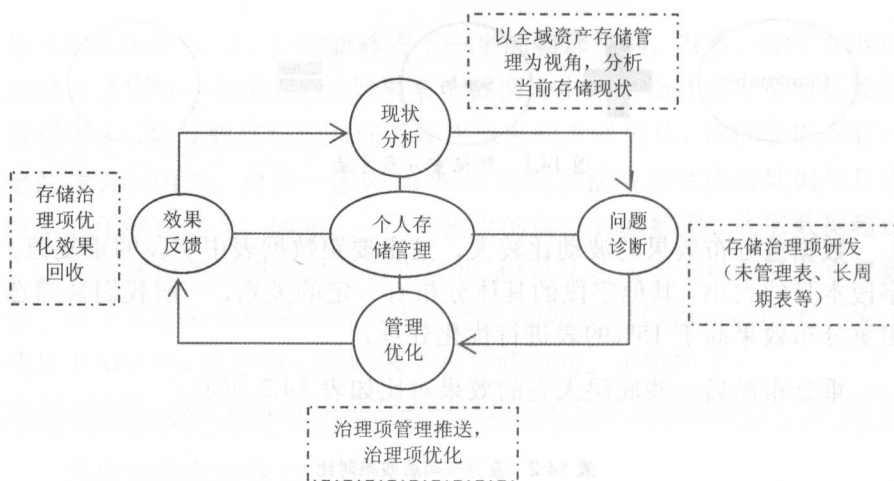


图 14.2 存储治理项优化主要流程图

## 14.4 生命周期管理

MaxCompute 作为阿里巴巴集团的大数据计算及服务引擎，存储着阿里系大量且非常重要的数据，从数据价值及数据使用性方面综合考虑，数据的生命周期管理是存储管理的一项重要手段。生命周期管理的根本目的就是用最少的存储成本来满足最大的业务需求，使数据价值最大化。

### 14.4.1 生命周期管理策略

#### 1. 周期性删除策略

所存储的数据都有一定的有效期，从数据创建开始到过时，可以周期性删除  $X$  天前的数据。例如对于 MySQL 业务库同步到 MaxCompute 的全量数据，或者 ETL 过程产生的结果数据，其中某些历史数据可能已经没有价值，且占用存储成本，那么针对无效的历史数据就可以进行

定期清理。

## 2. 彻底删除策略

无用表数据或者 ETL 过程产生的临时数据，以及不需要保留的数据，可以进行及时删除，包括删除元数据。

## 3. 永久保留策略

重要且不可恢复的底层数据和应用数据需要永久保留。比如底层交易的增量数据，出于存储成本与数据价值平衡的考虑，需要永久保留，用于历史数据的恢复与核查。

## 4. 极限存储策略

极限存储可以超高压缩重复镜像数据，通过平台化配置手段实现透明访问；缺点是对数据质量要求非常高，配置与维护成本比较高，建议一个分区有超过 5GB 的镜像数据（如商品维表、用户维表）就使用极限存储。

## 5. 冷数据管理策略

冷数据管理是永久保留策略的扩展。永久保留的数据需要迁移到冷数据中心进行永久保存，同时将 MaxCompute 中对应的数据删除。一般将重要且不可恢复的、占用存储空间大于 100TB，且访问频次较低的数据进行冷备，例如 3 年以上的日志数据。

## 6. 增量表 merge 全量表策略

对于某些特定的数据，极限存储在使用性与存储成本方面的优势不是很明显，需要改成增量同步与全量 merge 的方式，对于对应的 delta 增量表的保留策略，目前默认保留 93 天。例如，交易增量数据，使用订单创建日期或者订单结束日期作为分区，同时将未完结订单放在最大分区中，对于存储，一个订单在表里只保留一份；对于用户使用，通过分区条件就能查询某一段时间的数据。

## 14.4.2 通用的生命周期管理矩阵

随着业务的发展和不断的数据实践,我们慢慢摸索出一套适合大数据生命周期管理的规范,主要通过对历史数据的等级划分与对表类型的划分生成相应的生命周期管理矩阵。

### 1. 历史数据等级划分

目前我们对历史数据进行了重要等级的划分,主要将历史数据划分为 P0、P1、P2、P3 四个等级,其具体定义如下。

- P0: 非常重要的主题域数据和非常重要的应用数据,具有不可恢复性,如交易、日志、集团 KPI 数据、IPO 关联表。
- P1: 重要的业务数据和重要的应用数据,具有不可恢复性,如重要的业务产品数据。
- P2: 重要的业务数据和重要的应用数据,具有可恢复性,如交易线 ETL 产生的中间过程数据。
- P3: 不重要的业务数据和不重要的应用数据,具有可恢复性,如某些 SNS 产品报表。

### 2. 表类型划分

#### (1) 事件型流水表(增量表)

事件型流水表(增量表)指数据无重复或者无主键数据,如日志。

#### (2) 事件型镜像表(增量表)

事件型镜像表(增量表)指业务过程性数据,有主键,但是对于同样主键的属性会发生缓慢变化,如交易、订单状态与时间会根据业务发生变更。

#### (3) 维表

维表包括维度与维度属性数据,如用户表、商品表。

#### (4) Merge 全量表

Merge 全量表包括业务过程性数据或者维表数据。由于数据本身有

新增的或者发生状态变更,对于同样主键的数据可能会保留多份,因此可以对这些数据根据主键进行 Merge 操作,主键对应的属性只会保留最新状态,历史状态保留在前一天分区中。例如,用户表、交易表等都可以进行 Merge 操作。

#### (5) ETL 临时表

ETL 临时表是指 ETL 处理过程中产生的临时表数据,一般不建议保留,最多 7 天。

#### (6) TT 临时数据

TT 拉取的数据和 DbSync 产生的临时数据最终会流转到 ODS 层,ODS 层数据作为原始数据保留下来,从而使得 TT&DbSync 上游数据成为临时数据。这类数据不建议保留很长时间,生命周期默认设置为 93 天,可以根据实际情况适当减少保留天数。

#### (7) 普通全量表

很多小业务数据或者产品数据,BI 一般是直接全量拉取,这种方式效率高,对存储压力也不是很大,而且表保留很长时间,可以根据历史数据等级确定保留策略。

通过上述历史数据等级划分与表类型划分,生成相应的生命周期管理矩阵,如表 14.3 所示。

表 14.3 生命周期管理矩阵

		P0	P1	P2	P3
ODS 层	事件型流水表(增量表)	永久保留	3 年	365 天	180 天
	事件型镜像表(增量表)	永久保留	3 年	365 天	180 天
	维表(全量表)	33 天+极限存储	33 天+极限存储	33 天+极限存储	33 天+极限存储
	Merge 全量表	2 天	2 天	2 天	2 天
	普通全量表	3 年	365 天	365 天	180 天
	新同步全量表	3 天	3 天	3 天	3 天

续表

		P0	P1	P2	P3
DWD 层	事件型流水表（增量表）	永久保留	3 年	365 天	180 天
	事件型镜像表（增量表）	永久保留	3 年	365 天	180 天
	维表（全量表）	33 天+极限存储	33 天+极限存储	33 天+极限存储	33 天+极限存储
	普通全量表	3 年	365 天	365 天	180 天
DWS 层	各粒度数据	永久保留	3 年	3 年	3 年
临时存储区	ETL 临时表	7 天	3 天	3 天	3 天
	TT 临时数据	7 天	7 天	7 天	7 天
应用层	运营报表	永久保留	—	—	—
	对外数据	7 年	—	—	—
	内部产品	3 年	—	—	—

MaxCompute 集群中海量数据的存储和大量计算任务每天都会消耗巨额成本，并且随着数据量的不断增长，这个成本还在逐步增加。如何在服务好业务的前提下，更好地管控数据成本，提升资源利用率，已成为数据资产管理工作中非常重要的一环。

在阿里巴巴集团内部，大部分数据都会存储在 MaxCompute 集群上，数据以数据表的形式存在，并且数据表之间存在比较复杂的关联和上下游依赖关系。可以把数据表之间的依赖关系用树形结构形象化地表示，如图 14.3 所示。图中的 A、B、C 等代表不同的数据表，带箭头的连线代表数据表之间的依赖和关联关系。比如数据表 B、C、D 都依赖数据表 A，数据表 E 依赖数据表 B 和 C。

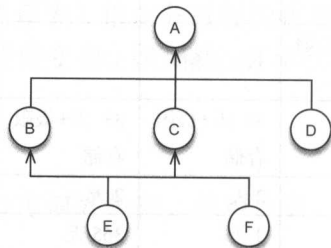


图 14.3 MaxCompute 数据表依赖关系示意图

MaxCompute 中的任何一个计算任务都会涉及计算和存储资源的消耗，其中计算资源的消耗主要考虑 CPU 消耗。为了下面更好地描述数据计量计费的算法和规则，特做如下定义：CPU 消耗的单位定义为 CU，代表 CPU 的一个核心（Core）运行一天的消耗量。存储资源的消耗主要考虑磁盘存储的消耗，这里采用国际通用的存储单位 PB 来衡量。例如：计算资源的单价为 1 元/CU，存储资源的单价为 1 元/PB 天。

## 14.5 数据成本计量

对数据成本的计量，可以采用最简单的方式，将一个数据表的成本分为存储成本和计算成本。存储成本是为了计量数据表消耗的存储资源，计算成本是为了计量数据计算过程中的 CPU 消耗。但是，对这样的数据成本计量方式会存在较大的质疑和挑战。例如，如图 14.4 所示，表 D 是业务方的一个数据表，表 D 依赖表 C，但是为了产生表 C，往往上面存在一个较长的数据刷新链路。表 C 的成本可能是 10 元，但是表 A、B 可能会是 100 元。像这样的情况，如果表 C 的成本仅仅用数据表 C 自身的存储和计算成本衡量显然是不合理、不准确的。



图 14.4 数据刷新链路示意图



因此，在计量数据表的成本时，除考虑数据表本身的计算成本、存储成本外，还要考虑对上游数据表的扫描带来的扫描成本。我们将数据成本定义为存储成本、计算成本和扫描成本三个部分。

通过在数据成本计量中引入扫描成本的概念，可以避免仅仅将表自身硬件资源的消耗作为数据表的成本，以及对数据表成本进行分析时，孤立地分析单独的一个数据表，能够很好地体现出数据在加工链路中的上下游依赖关系，使得成本的评估尽量准确、公平、合理。

## 14.6 数据使用计费

在上一节中，已经清楚地将数据成本分为：存储成本、计算成本和扫描成本。那么对于数据表的使用计费，在阿里巴巴集团内部，分别依据这三部分成本进行收费，称为：计算付费、存储付费和扫描付费。

我们把数据资产的成本管理分为数据成本计量和数据使用计费两个步骤。通过成本计量，可以比较合理地评估出数据加工链路中的成本，从成本的角度反映出在数据加工链路中是否存在加工复杂、链路过长、依赖不合理等问题，间接辅助数据模型优化，提升数据整合效率；通过数据使用计费，可以规范下游用户的数据使用方法，提升数据使用效率，从而为业务提供优质的数据服务。

# 第 15 章

## 数据质量

随着 IT 向 DT 时代的转变，数据的重要性不言而喻，数据的应用也日趋繁茂，数据正扮演着一个极其重要的角色。而对于被日益重视的数据，如何保障其质量也是阿里巴巴乃至业界都普遍关注的一个话题。本章将介绍阿里巴巴如何保障数据仓库的数据质量。

数据质量是数据分析结论有效性和准确性的基础，也是这一切的前提。如何保障数据质量，确保数据可用性是阿里巴巴数据仓库建设不容忽视的环节。接下来将通过数据质量原则逐一展开介绍阿里巴巴对数据仓库数据质量建设的方法。

### 15.1 数据质量保障原则

如何评估数据质量的好坏，业界有不同的标准，而阿里巴巴对数据仓库主要从四个方面进行评估，即完整性、准确性、一致性和及时性，如图 15.1 所示。



图 15.1 数据质量保障原则

### 1. 完整性

完整性是指数据的记录和信息是否完整，是否存在缺失的情况。数据的缺失主要包括记录的缺失和记录中某个字段信息的缺失，两者都会造成统计结果不准确，所以说完整性是数据质量最基础的保障。比如交易中每天支付订单数都在 100 万笔左右，如果某天支付订单数突然下降到 1 万笔，那么很可能就是记录缺失了。对于记录中某个字段信息的缺失，比如订单的商品 ID、卖家 ID 都是必然存在的，这些字段的空值个数肯定是 0，一旦大于 0 就必然违背了完整性约束。

### 2. 准确性

准确性是指数据中记录的信息和数据是否准确，是否存在异常或者错误的信息。比如一笔订单如果出现确认收货金额为负值，或者下单时间在公司成立之前，或者订单没有买家信息等，这些必然都是有问题的。如何确保记录的准确性，也是保障数据质量必不可少的一个原则。

### 3. 一致性

一致性一般体现在跨度很大的数据仓库体系中，比如阿里巴巴数据仓库，内部有很多业务数据仓库分支，对于同一份数据，必须保证一致性。例如用户 ID，从在线业务库加工到数据仓库，再到各个消费节点，

必须都是同一种类型，长度也需要保持一致。所以，在建设阿里巴巴数据仓库时，才有了公共层的加工，以确保数据的一致性。

#### 4. 及时性

在确保数据的完整性、准确性和一致性后，接下来就要保障数据能够及时产出，这样才能体现数据的价值。一般决策支持分析师都希望当天就能够看到前一天的数据，而不是等三五天才能看到某一个数据分析结果；否则就已经失去了数据及时性的价值，分析工作变得毫无意义。现在对时间要求更高了，越来越多的应用都希望数据是小时级别或者实时级别的。比如阿里巴巴“双 11”的交易大屏数据，就做到了秒级，及时性同样是保障数据质量的一个重要原则。

## 15.2 数据质量方法概述

在阿里巴巴数据仓库建设过程中，经过不断的实践，慢慢摸索出一套适合大数据的数据质量方法，在满足以上四个原则的基础上，为阿里巴巴数据做基础保障。

阿里巴巴业务复杂，种类繁多的产品每天产生数以亿计的数据，每天的数据量都在 PB 级以上，而数据消费端的应用又层出不穷，各类数据产品如雨后春笋般出现。为了不断满足这些数据应用的需要，数据仓库的规模在不断膨胀，同时数据质量的保障也越来越复杂。基于这些背景，我们提出了一套数据质量建设方法，如图 15.2 所示。

这套方法主要包括如下几个方面。

### 1. 消费场景知晓

消费场景知晓部分主要通过数据资产等级和基于元数据的应用链路分析解决消费场景知晓的问题。根据应用的影响程度，确定资产等级；

根据数据链路血缘，将资产等级上推至各数据生产加工的各个环节，确定链路上所涉及数据的资产等级和在各加工环节上根据资产等级的不同所采取的不同处理方式。



图 15.2 数据质量建设方法

## 2. 数据生产加工各个环节卡点校验

数据生产加工各个环节卡点校验部分主要包括在线系统和离线系统数据生产加工各个环节的卡点校验。其中在线系统指 OLTP (On-Line Transaction Processing, 联机事务处理) 系统; 离线系统指 OLAP (On-Line Analytical Processing, 联机分析处理) 系统。

在线系统生产加工各环节卡点校验主要包括两个方面：根据资产等级的不同，当对应的业务系统变更时，决定是否将变更通知下游；对于高资产等级的业务，当出现新业务数据时，是否纳入统计中，需要卡点审批。

离线系统生产加工各环节卡点校验主要包括代码开发、测试、发布和历史或错误数据回刷等环节的卡点校验，针对数据资产等级的不同，对校验的要求有所不同。

### 3. 风险点监控

风险点监控部分主要是针对在数据日常运行过程中可能出现的数据质量和时效等问题进行监控，包括在线数据和离线数据的风险点监控两个方面。在线数据的风险点监控主要是针对在线系统日常运行产出的数据进行业务规则的校验，以保证数据质量，其主要使用实时业务检测平台 BCP (Biz Check Platform)；离线数据的风险点监控主要是针对离线系统日常运行产出的数据进行数据质量监控和时效性监控，其中数据质量监控主要使用 DQC，时效性监控主要使用摩萨德。

### 4. 质量衡量

对质量的衡量既有事前的衡量，如 DQC 覆盖率；又有事后的衡量，主要用于跟进质量问题，确定质量问题原因、责任人、解决情况等，并用于数据质量的复盘，避免类似事件再次发生。根据质量问题对不同等级资产的影响程度，确定其是属于低影响的事件还是具有较大影响的故障。质量分则是综合事前和事后的衡量数据进行打分。

### 5. 质量配套工具

针对数据质量的各个方面，都有相关的工具进行保证，以提高效能。接下来将对数据质量保障的各个方面进行介绍。

## 15.2.1 消费场景知晓

在数据快速增长的情况下，数据类产品和日常决策支持系统也应运

而生，且层出不穷，而对于数据不断增加的需求，数据仓库已经应接不暇，数据研发工程师已经难以确认几百 PB 的数据到底是否都是重要的，是否都要进行保障，是否有一些数据已经过期了，是否所有需要都要精确地进行质量保障，上述数据质量保障的四个原则是否还适用于所有的数据……这些疑问伴随着每一个数据开发人员。数据规模的膨胀考验数据质量的原则，同时也是对数据加工的考验。基于不断膨胀的数据和对数据类的应用，阿里巴巴内部提出了数据资产等级的方案，旨在解决消费场景知晓的问题。

### 1. 数据资产等级定义

什么是数据资产等级？针对阿里巴巴庞大的数据仓库，数据的规模已经达到 EB 级别，对于这么大的数据量，如果一概而论势必会造成精力无法集中、保障无法精确，因此给数据划分等级势在必行。经过不断地讨论和研究，最终将数据分为五个等级，即毁灭性质、全局性质、局部性质、一般性质和未知性质，不同性质的重要性依次降低，具体定义如下。

- 毁灭性质，即数据一旦出错，将会引起重大资产损失，面临重大收益损失，造成重大公关风险。
- 全局性质，即数据直接或者间接用于集团级业务和效果的评估、重要平台的运维、对外数据产品的透露、影响用户在阿里系网站的行为等。
- 局部性质，即数据直接或间接用于内部一般数据产品或者运营/产品报告，如果出现问题会给事业部或业务线造成影响，或者造成工作效率损失。
- 一般性质，即数据主要用于小二的日常数据分析，出现问题几乎不会带来影响或者带来的影响极小。
- 未知性质，不能明确说出数据的应用场景，则标注为未知。

对于不同的数据资产等级使用英文 Asset 进行标记，毁灭性质标记为 A1 等级，全局性质标记为 A2 等级，局部性质标记为 A3 等级，一般性质标记为 A4 等级，未知性质则标记为 Ax 等级。在重要程度上，

$A1 > A2 > A3 > A4 > Ax$ 。另外，如果一份数据出现在多个应用场景中，则遵循就高原则。

## 2. 数据资产等级落地方法

前面已经给出了数据资产等级的定义，但是对于如此庞大的数据量，如何给每一份数据都打上一个等级标签呢？这里首先介绍数据的简单流转过程。

数据是从业务系统中产生的，经过同步工具进入数据仓库系统中，在数据仓库中进行一般意义上的清洗、加工、整合、算法、模型等一系列运算后，再通过同步工具输出到数据产品中进行消费。而从业务系统到数据仓库再到数据产品都是以表的形式体现的，其流转过程如图 15.3 所示。

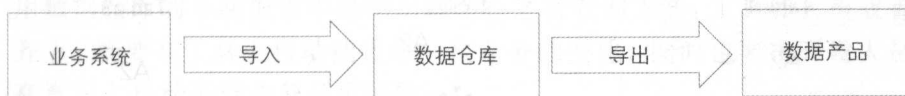


图 15.3 数据流转过程

同步到数据仓库（对应到阿里巴巴就是 MaxCompute 平台）中的都是业务数据库的原始表，这些表主要用于承载业务需求，往往不能直接用在数据产品中，在数据产品中使用的都是经过数据仓库加工后的产出表。

有了数据产品或者数据应用的概念，同时也知道了哪些表是为哪个数据产品或者应用服务的，就可以借助强大的元数据知道整个数据仓库中的哪些表服务于这个数据产品，因此通过给不同的数据产品或者应用划分数据资产等级，再依托元数据的上下游血缘，就可以将整个消费链路打上某一类数据资产的标签，这样就可以将数以亿计的数据进行分类了。关于元数据的加工计算详见第 12 章“元数据”。

这里以阿里巴巴生意参谋产品（见“数据应用篇”介绍）为例，简单介绍数据资产等级打标的过程。生意参谋是一款为商家提供服务的数数据类产品，完全依托数据，为商家进行决策支持。它每天零点开始同步，计算前一天的数据，8:00 给到商家，提供服务。产品每一个页面的每一



个模块基本上都是通过数据表输出展现的,不同模块数据的重要等级也就决定了相关表的重要等级,决定了这个导出表的重要等级。比如生意参谋为 A2 等级的业务,那么对应这个导出表的资产等级就是 A2,所有加工这个表的上游链路上的所有表都将会打上 A2 资产等级的标签,同时会标注为生意参谋产品使用。如图 15.4 所示,生意参谋打上了 A2 的标记,直接服务于生意参谋的表 Table1、Table2、Table3 进行 A2-生意参谋标记,根据血缘上溯,这三个表的上游都将打上 A2 的标记,一直标记到前台业务系统,将血缘贯通。

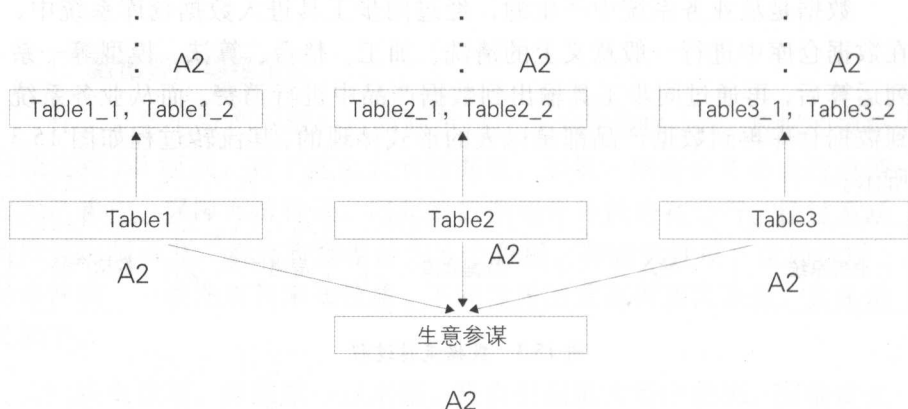


图 15.4 生意参谋产品打标过程

通过如上步骤,就完成了数据资产等级的确认,给不同的数据定义了不同的重要程度,当然这里是需要元数据支撑的。

解决了消费场景知晓的问题,就知道了数据的重要等级,针对不同的等级,也将采取不同的保障措施。接下来将介绍阿里巴巴在数据仓库中针对不同资产等级的数据的保障方法。

## 15.2.2 数据加工过程卡点校验

### 1. 在线系统卡点校验

在线系统数据加工过程卡点校验,主要是指在在线业务系统的数据

生成过程中进行的卡点校验。阿里巴巴的 OLTP 系统比较丰富,也比较复杂,比如交易、会员、商品、营销、评价、退款、客服等,这些服务于用户购物的在线业务系统,既满足了用户日常需求,也是数据仓库的数据来源,因此既要保障数据的准确性,同时也要保障和离线数据的一致性。

关于对数据准确性的保障,主要以数据监控为主,在 15.2.3 节“风险监控”中进行介绍。本节主要介绍在线数据和离线数据一致性的保障问题。

在线业务复杂多变,总是在不断地变更,每一次变更都会带来数据的变化,数据仓库需要适应这多变的业务发展,及时做到数据的准确性。基于此,在线业务的变更如何高效地通知到离线数据仓库,同样也是需要考虑的问题。为了保障在线数据和离线数据的一致性,阿里巴巴在使用数据仓库的不断摸索中总结出两个行之有效的方法:工具和人员双管齐下。既要在工具上自动捕捉每一次业务的变化,同时也要求开发人员在意识上自动进行业务变更通知。

工具,首先是发布平台。在业务进行重大变更时,订阅这个发布过程,然后给到离线开发人员,使其知晓此次变更的内容。业务系统繁杂,日常发布变更数不胜数,如果每一次变更都要知会离线业务,那势必会造成不必要的浪费,同时会影响在线业务迭代的效率。因此这里充分发挥了数据资产等级的作用,针对全集团重要的高等级数据资产,整理出哪些变更会影响数据的加工。比如财报,这个自然是 A1 等级的资产,如果业务系统的改造会影响财报的计算,如约定好的计算口径被业务系统发布变更修改了,那么务必要告知离线业务,作为离线开发人员也必须主动关注这类发布变更信息。所以发布平台集成了通知功能,针对重要的场景发布会进行卡点,确认通知后才能完成发布。

其次是数据库表的变化感知。无论是随着业务发展而做的数据库扩容还是表的 DDL 变化,都需要主动通知到离线开发人员。数据仓库在进行数据抽取时,采用的是 DataX 工具,可能限制了某个数据库表,如果发生数据库扩容或者迁移,DataX 工具是感知不到的,结果可能会导致数据抽取错漏,影响一系列的下游应用。对此,阿里巴巴是通过数

据库平台进行库表变更通知发送的。

有了好的工具的辅助，而操作工具的开发人员更是核心。数据资产等级的上下游打通，同样也要将这个过程给到在线开发人员，使其知晓哪些是重要的核心数据资产，哪些暂时还只是作为内部分析数据使用。要提高在线开发人员的意识，通过培训，将离线数据的诉求、离线数据的加工过程、数据产品的应用方式告诉在线业务开发人员，使其意识到数据的重要性，了解数据的价值，同时也告知出错后果，使在线开发人员在完成业务目标时，也要注重数据的目标，做到业务端和数据端一致。

阿里巴巴的数据仓库通过这种机制，在很大程度上保障了第一手数据的准确性。但是业务、技术都在不断快速地发展着，我们也在不断地摸索更高效、更准确的在线业务保障方案，为在线数据质量保驾护航。

## 2. 离线系统卡点校验

前文已有介绍，数据从在线业务系统到数据仓库再到数据产品的过程中，需要在数据仓库这一层完成数据的清洗、加工。正是有了数据的加工，才有了数据仓库模型和数据仓库代码的建设。如何保障数据加工过程中的质量，是离线数据仓库保障数据质量的一个重要环节。

首先是代码提交时的卡点校验。数据研发人员素质不同，代码能力也有差异，代码质量也就难以得到高效保障。在此背景下，我们上线了代码扫描工具 SQLSCAN，针对每一次提交上线的代码进行扫描，将风险点提示出来。具体规则已经在第4章“离线数据开发”中进行了介绍。

其次是任务发布上线时的卡点校验。为了保障线上数据的准确性，每一次变更都需要线下完成测试后再发布到线上环境中，线上测试通过后才算发布成功。发布上线前的测试主要包括 Code Review 和回归测试，对于资产等级较高的任务变更发布，则采取强阻塞的形式，必须通过在彼岸完成回归测试之后才允许发布。回归测试一方面要保证新逻辑的正确；另一方面要保证不影响非此次变更的逻辑。发布上线后可以在线上做 Dry Run 测试或真实环境运行测试，其中 Dry Run 测试，不执行代码，仅运行执行计划，避免线上和线下环境不一致导致语法错误；真实环境的运行测试，则使用真实数据进行测试。

最后是节点变更或数据重刷前的变更通知。一般建议使用通知中心的将变更原因、变更逻辑、变更测试报告和变更时间等自动通知下游，下游对此次变更没有异议后，再按照约定时间执行发布变更，将变更对下游的影响降至最低。

### 15.2.3 风险点监控

风险点监控主要是针对数据在日常运行过程中容易出现的风险进行监控并设置报警机制，主要包括在线数据和离线数据运行风险点监控。

#### 1. 在线数据风险点监控

在线业务系统的数据生产过程需要保证数据质量，主要根据业务规则对数据进行监控。阿里巴巴主要采用实时业务检测平台 BCP，用于保障在线系统的数据质量。

BCP 针对数据库表的记录进行规则校验。在每一个业务系统中，当完成业务过程进行数据落库时，BCP 同时订阅一份相同的数据，在 BCP 系统中进行逻辑校验，当校验不通过时，以报警的形式披露出来给到规则订阅人，以完成数据的校对。所以，采用 BCP 进行校验的过程是，首先，用户在 BCP 平台进行数据源订阅，以获取需要校验的数据源；然后，针对所订阅的数据源进行规则的编写，即校验的逻辑，这些规则是至关重要的，也是校验的核心，只要这些规则通过了，即认为这条记录是对的；最后，配置告警，针对不同的规则配置不同的告警形式。有了这样一套流程，就能够在第一时间发现脏数据并通知到订阅人，既减少了用户数据错误的投诉，也减少了离线数据错误的回滚。BCP 在很大程度上减少了脏数据，为数据的准确性把了第一道关。

比如交易系统配置的一些监控规则，如订单拍下时间、订单完结时间、订单支付金额、订单状态流转等都配置了校验规则。订单拍下时间肯定不会大于当天时间，也不会小于淘宝创立时间，一旦出现异常的订单创建时间，就会立刻报警，同时报警给到多人。通过这种机制，可以

及时发现并解决问题。BCP 的配置和运行成本较高，主要根据数据资产等级进行监控。

## 2. 离线数据风险点监控

离线数据风险点监控主要包括对数据准确性和数据产出及时性的监控。

### (1) 数据准确性

数据准确性是数据质量的关键，因此数据准确成为数据质量的重中之重，是所有离线系统加工时的第一保障要素。阿里巴巴主要使用 DQC 来保障数据的准确性，其具体功能说明和规则配置可以参考第 4 章“离线数据开发”中的内容。

DQC 检查其实也是运行 SQL 任务，只是这个任务是嵌套在主任务中的，一旦检查点太多自然就会影响整体的性能，因此还是依赖数据资产等级来确定规则的配置情况。比如 A1、A2 类数据监控率要达到 90% 以上，规则类型需要三种及以上，而不重要的数据资产则不强制要求。

类似的规则都是由离线开发人员进行配置来确保数据准确性的。当然不同的业务还是会有业务规则的约束，这些规则来源于数据产品或者说消费的业务需求，由消费节点进行配置，然后上推到离线系统的起点进行监控，做到规则影响最小化。

### (2) 数据及时性

在确保数据准确性的前提下，需要进一步让数据能够及时地提供服务；否则数据的价值将大幅度降低，甚至没有价值，所以确保数据及时性也是保障数据质量重中之重的一环。

对于阿里巴巴大部分离线任务，一般是以天作为时间间隔的。对于天任务，数据产品或者管理层决策报表一般都要求在每天 9:00 甚至更早的时间产出。为确保前一天的数据完整，天任务是从零点开始运行的。由于计算加工的任务都是在夜里运行的，而要确保每天的数据能够按时产出，则需要进行一系列的报警和优先级设置，使得重要的任务优先且正确产出。这里的重要性即前面所述的数据资产等级，资产等级高的业

务必定优先保障。

①任务优先级。如何确保重要任务获得高优先级，是调度平台需要重点考虑的问题。如前文所述，调度是一个树形结构，当配置了叶子节点的优先级后，这个优先级会传递到所有上游节点，所以优先级的设置都是给到叶子节点，而叶子节点往往就是服务业务的消费节点。因此在优先级的设置上，首先是确定业务的资产等级，等级高的业务所对应的消费节点自然配置高优先级，一般业务则对应低优先级，确保高等级业务准时产出。

②任务报警。任务报警和优先级类似，也是通过叶子节点传递。任务在运行过程中难免会出错，因此要确保任务能够高效、平稳地执行，需要有一个监控报警系统，对于高优先级的任务，一旦发现任务出错或者可能出现产出延迟，就要报警给到任务和业务 Owner。阿里巴巴使用自主开发的监控报警系统——摩萨德来监控任务的实时运行状况，若发现异常则执行不同等级的报警，根据不同的资产等级执行强保障或弱保障。高资产等级的业务才会获得强保障，任务出错或可能延迟则执行电话报警；一般业务只会做到短信或者邮件告警。

③摩萨德。摩萨德是离线任务的监控报警系统，它会根据离线任务的运行情况实时决策是否告警、何时告警、告警方式、告警给谁等。摩萨德经过几年的发展，目前已经比较成熟，是数据运维不可或缺的保障工具。摩萨德提供了两个最主要的功能：强保障监控和自定义告警。

强保障监控是摩萨德的核心功能，也是紧紧围绕运维目标即业务保障而设计的，只要在业务的预警时间受到威胁，摩萨德就一定会告警出来给到相关人员。强保障监控主要包括：

- 监控范围——设置了强保障业务的任务及其上游所有的任务都会被监控。
- 监控的异常——任务出错、任务变慢、预警业务延迟。
- 告警对象——默认是任务 Owner，也可以设置值班表到某一个人。
- 何时告警——根据业务设置的预警时间判断何时告警。
- 告警方式——根据任务的重要紧急程度，支持电话、短信、旺旺、邮件告警。



比如生意参谋业务，定义的数据资产等级是 A2，要求早上 9:00 产出数据给到商家，因此我们给生意参谋业务定义一个强保障监控，业务产出时间是 9:00，业务预警时间是 7:00。这里的预警时间是指一旦摩萨德监控到当前业务的产出时间超出预警时间时就会打电话给到值班人员进行预警，比如摩萨德推测生意参谋的产出时间要到 7:30，那么电话告警就出来了，由值班人员来判断如何加速产出。那么是怎样判断当前执行会超过预警时间的呢？摩萨德是根据当前业务上所有任务最近 7 天运行的平均时间来推算的，虽然有误判的可能性，但是总体还是非常准的，可以接受。这个是预警判断，也就是产出延迟判断。

另外还有出错报警，当重要业务比如生意参谋产出路径上有个任务运行出错了，此时摩萨德依然会根据预警时间判断要不要马上报警给到值班人员。这里注意，出错报警的时机也是根据产出预警时间来判断的，对于生意参谋而言，夜里一旦任务出错肯定是要立即报警的，因为产出要求早，容不得推迟；而对于不重要的业务即资产等级低的业务或者产出时间要求晚的业务，摩萨德则会根据预警时间来判断合适的报警时间点，比如算法类的业务需求，可能当天产出即可，那么摩萨德会根据预警时间，将出错报警时间推迟到 9:00 以后，即上班了再报警，以减轻值班人员的夜里负担。

除了针对业务的强保障监控，还有自定义监控。自定义监控是摩萨德比较轻量的监控功能，用户可以根据自己的需要进行配置，主要包括：

- 出错告警——可根据应用、业务、任务三个监控对象进行出错告警配置，监控对象出错即告警给到人/Owner/值班表。
- 完成告警——可根据应用、业务、任务三个监控对象进行完成情况告警配置，监控对象完成即告警给到人/Owner/值班表。
- 未完成告警——可根据应用、业务、任务三个监控对象进行未完成情况告警配置，监控对象未完成即告警给到人/Owner/值班表。
- 周期性告警——针对某个周期的小时任务，如果在某个时间未完成，即告警给到人/Owner/值班表。
- 超时告警——根据任务运行时长进行超时告警配置，任务运行超过指定时间即告警给到人/Owner/值班表。

有时候是非重要业务的某个任务，但任务 Owner 还是想看一下产出情况，此时就可以设置自定义告警。比如生意参谋业务，因为预警时间都是定在业务上的，如果其中某个表 Owner 希望每天都能监控产出时间，此时就可以自定义一个监控，监控这个任务的产出时间，当然如果 Owner 认为这个表一定要 2:00 产出，2:00 没有产出则电话告警，也是可以的，灵活性比较大。

另外，摩萨德提供了甘特图的服务，针对业务的运行情况，摩萨德会提供一条关键路径，即完成业务的最慢任务链路图。因为每个业务上游可能有成千上万个任务，所以这条关键路径对于业务链路优化来说非常重要。

通过这一系列的报警监控机制，摩萨德能够很好地服务于业务产出，保障业务产出的及时性。

## 15.2.4 质量衡量

前面章节中已经给出了阿里巴巴对保障数据仓库数据质量的各种方案，但是如何评价这些方案的优劣，需要一套度量指标。

### 1. 数据质量起夜率

前文在介绍数据及时性时已经提到，数据产品或者管理层决策日报一般都要求在上午 9:00 之前提供，数据仓库的作业任务都是在凌晨运行的，一旦数据出现问题就需要开发人员起夜进行处理。因此，每个月的起夜次数将是衡量数据质量建设完善度的一个关键指标。如果频繁起夜，则说明数据质量的建设不够完善，所以在阿里巴巴数据仓库数据质量度量体系里，起夜率是一个首先要考虑的指标。

对于数据质量本身，将通过数据质量事件和数据质量故障来衡量。

### 2. 数据质量事件

针对每一个数据质量问题，都记录一个数据质量事件。



数据质量事件，首先，用来跟进数据质量问题的处理过程；其次，用来归纳分析数据质量原因；第三，根据数据质量原因来查缺补漏，既要找到数据出现问题的原因，也要针对类似问题给出后续预防方案。

因此，数据质量事件既用来衡量数据本身的质量，也用来衡量数据链路上下游的质量，是数据质量的一个重要度量指针。

### 3. 数据质量故障体系

对于严重的数据质量事件，将升级为故障。故障，是指问题造成的影响比较严重，已经给公司带来资产损失或者公关风险。比如财报计算错误、卖家结算数据错误、微贷信用数据错误、高管报表错误或者延迟等都将带来恶劣的影响。此类数据质量问题，已经不仅仅是一个事件，而是升级为故障。当然，数据质量故障对于开发人员和部门来讲，都是一个重要考核点，因此也是数据质量度量最严的一个指标。

数据从采集到最后的消费，整个链路要经过几十个系统，任何一个环节出现问题，都会影响数据的产出，因此需要一种机制，能够将各团队绑在一起，目标一致，形成合力，故障体系在这个背景下应运而生。一旦出现故障，就会通过故障体系，要求相关团队第一时间跟进解决问题，消除影响。

#### (1) 故障定义

首先识别出重要的业务数据，并注册到系统中，填写相关的业务情况，如技术负责人、业务负责人、数据应用场景、延迟或错误带来的影响、是否会发生资产损失等，完成后，会将这部分数据的任务挂到平台基线上，一旦延迟或错误即自动生成故障单，形成故障。

#### (2) 故障等级

故障发生后，会根据一定的标准判断故障等级，如故障时长、客户投诉量、资金损失等，将故障按 p1~p4 定级，各团队会有故障分的概念，到年底会根据故障分情况来判断本年度的运维效果。

#### (3) 故障处理

故障发生后，需要快速地识别故障原因，并迅速解决，消除影响。

在处理故障的过程中，会尽快将故障的处理进度通知到相关方，尽可能减少对业务的影响。

#### (4) 故障 Review

对于故障会进行 Review，即分析故障的原因、处理过程的复盘、形成后续解决的 Action，并且都会以文字的形式详细记录，对故障的责任进行归属，一般会到具体的责任人。注意，对故障责任的判定，不是为了惩罚个人，而是通过对故障的复盘形成解决方案，避免问题再次发生。

## 第4篇

# 数据应用篇

### • 第16章 数据应用

## 第16章

# 数据应用

全球知名咨询公司麦肯锡称：“数据，已经渗透到当今每一个行业和业务职能领域，成为重要的生产要素。人们对于海量数据的挖掘和运用，预示着新一波生产率增长和消费者盈余浪潮的到来。”

本书在前面的章节中已经深入介绍了大数据建设与管理的方法论和实践，“生产要素”已经准备好，需要通过合适的方式提供给不同类型的用户，让数据最大化地发挥价值。阿里巴巴作为一家天然的大数据公司，对数据的应用表现在各个方面，如搜索、推荐、广告、金融、信用、保险、文娱、物流等业务。将数据提供给商家，可以用于指导商家的数据化运营，为商家提供多样化、普惠性的数据赋能；将数据提供给阿里巴巴内部的搜索、推荐、广告、金融等平台，可以用于实现更好的搜索体验、更精准的个性化推荐，优化购物体验，更精准地进行广告投放、更普惠的金融服务等；将数据提供给阿里巴巴内部员工，如客服、运营、产品经理和管理人员等，可以用于数据化运营和决策；同时，阿里巴巴也将多年来的数据技术能力对外赋能，ISV、研究机构和社会组织可以利用阿里巴巴开放的数据能力和技术，进行经济发展、国内外打假、工业制造等议题的研究和实践，从而促进社会经济、民生的发展等。

本章主要介绍两个应用：提供给外部商家使用的数据产品平台——生意参谋和服务于阿里巴巴内部的数据产品平台。

## 16.1 生意参谋

作为大数据公司，阿里巴巴在推动业务数据化的同时，也在不断地帮助商家实现数据业务化。在对外产品方面，阿里巴巴主要以“生意参谋”作为官方统一的数据产品平台，为商家提供多样化、普惠性的数据赋能。

截至 2016 财年，生意参谋累计服务商家已超 2000 万，月服务商家超 500 万。在月成交额 30 万元以上的商家中，逾 90% 在使用生意参谋；在月成交额 100 万元以上的商家中，逾 90% 每月登录天次超 20 次。

本节主要介绍生意参谋是什么，它主要为阿里生态下的商家提供哪些数据服务，以及随着 DT 时代全面来临，它将如何驱动商家重视数据化运营，进而实现“数据赋能商家”这一重要理念。

### 16.1.1 背景概述

生意参谋诞生于 2011 年，最早是应用于阿里巴巴 B2B 市场的数据工具，2013 年 10 月才正式进入淘系。

当时阿里淘系的数据产品曾一度多达 38 个，不同产品的统计方式不同，相同指标在不同产品中的数据也有所差异，这给商家带来不少的困扰。

为了保证用户体验，从 2014 年起，依托阿里巴巴内部的 OneData 体系建设的、在数据一致性方面更具优势的生意参谋陆续整合量子恒道、数据魔方等其他数据产品，并在 2015 年年底升级为官方统一的商家数据产品平台。由此，商家只要通过生意参谋一个平台，就能体验统一、稳定、准确的官方数据服务。

当然，长达两年的整合升级并不是简单地对多个数据产品进行功能整合，而是在保留其核心功能的同时，对其进行优化，同时不断拓展新平台的服务能力和服务范围。

在整合量子恒道时，同步推出大促活动看板、实时直播大屏、自助

取数等重要功能；在整合数据魔方时，推出功能定位相似的“市场行情”，还同步上线数据作战室这款风靡众多高端商家的大促直播利器……

经过不断拓展，目前平台的数据已覆盖淘宝、天猫等阿里系所有平台和 PC、无线等终端，涉及指标上千个。在产品功能方面，已拥有店铺自有分析、店铺行业分析、店铺竞争分析三大基础业务模块。另外，它还支持多个专题工具的使用和自助取数等个性化需求。

生意参谋产品理念如图 16.1 所示。

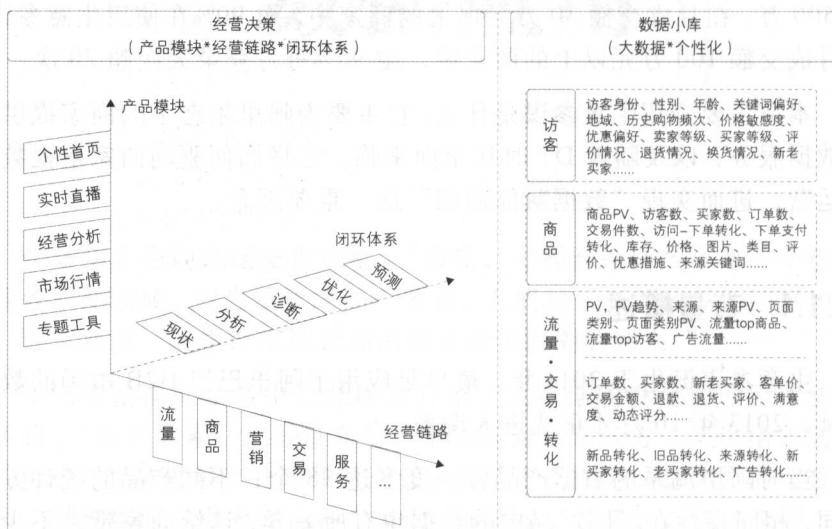


图 16.1 生意参谋产品理念图

2016 年，为进一步满足商家数据需求，阿里巴巴在门户、数据内容、产品形态三个方面对生意参谋进行全新升级。

在门户方面，首页支持多岗多面、多店融合，商家可根据不同岗位需求，选择页面中出现哪些数据；在数据内容方面，加强商家中后台数据突破，新增服务、物流等环节的数据服务，进一步满足商家全渠道、全链路的数据需求；在产品形态方面，更注重深度分析、诊断、建议、优化和预测。

随着网红经济的爆发，这一年，生意参谋还尝试了布局电商外领域

——与新浪微博、优酷等自媒体平台合作，推出生意参谋 CP 版（含微博版、优酷版）。网红或达人通过生意参谋，可了解自己的内容影响力、粉丝用户画像、商家合作效果，商家也可通过平台了解网红引流效果，从而更好地制定推广策略。

16.1.2 功能架构与技术能力

如上所述，生意参谋发展到今天，已经不是一个简单的数据产品或数据工具，而是集合了此前集团的多个数据产品，同时包含数据作战室、市场行情、装修分析、流量纵横、竞争情报等多个产品在内的数据产品平台。

目前平台共有七个板块，除首页外，还有实时直播、经营分析、市场行情、自助取数、专题工具、数据学院，如图 16.2 所示。不同板块的数据不尽相同，但又彼此联系。从商家实际应用场景来看，这些数据服务可以划分为三个维度，即看我情、看行情和看敌情。

首页	专属用户的个性化首页，可定制；常用功能模块聚合入口
实时直播	以店铺实时动态数据为切入点，提供实时数据的查询与分析；如数据作战室
经营分析	结合大环境，对经营全链路的各个环节进行分析、诊断等；如装修分析等
市场行情	以行业分析、竞争情况为切入点，对市场动态进行分析
自助取数	提供数据定制、查询、导出等高端数据服务、灵活可配置，周期可定制
专题工具	着重专题分析和一站式优化工具，含竞争情报、流量纵横等
数据学院	教学相长，可帮助商家快速提升数据化运营能力

图 16.2 生意参谋功能架构图

1. 看我情

不管是哪一层级的商家，看数据一般都是优先关注自身店铺。如果

连自身数据都不关注，那么了解再多的行业数据和对手数据也无济于事。在一定程度上，分析“我情”是店铺数据化运营的根本。

在生意参谋上，“我情”的数据主要基于店铺经营全链路的各个环节进行设计。以“经营分析”为例，这个板块依次为商家提供流量、商品、交易、服务、物流、营销等不同环节的数据服务，不同服务还能再往下细分，如在“流量分析”下，还会再提供流量地图、访客分析、装修分析等更细粒度的数据。基本上，一个访客从未进店到进店，再到店内流转，最终交易转化，转化后的评价和物流情况都可以通过“经营分析”一站式获取。

## 2. 看行情

在线上零售环境竞争程度还不十分激烈的时候，店铺埋头苦干，修好内功，或许就能独辟蹊径，脱颖而出。但现在，随着线上线下不断融合，有实力者不断入局，线上竞争日益加剧。在这个过程中，店铺要想运营好，就不得不经常关注行业动态了。只有知道外界在关注什么、在发生什么变化，才有可能把握市场动态，挖掘先机。

基于此，生意参谋通过市场行情，为商家提供了行业维度的大盘分析，包括行业洞察、搜索词分析、人群画像等。其中，行业洞察可以从品牌、产品、属性、商品、店铺等粒度对本行业数据进行分析；通过搜索词分析可以了解不同关键词的近日表现，从而反推消费者偏好；人群画像能从人群维度入手，直接提供买家人群、卖家人群、搜索人群三大人群的数据洞察信息。

## 3. 看敌情

大家都知道，商场如战场，只有知己知彼，才能百战百胜，因此关注“敌情”十分重要，这也是很多店铺发展到一定阶段后的迫切需求。

但是，竞争对手的数据十分敏感，生意参谋的产品设计原则之一又是确保商家数据安全，要如何权衡两者的关系呢？我们的解决方案是推出“竞争情报”这一专题工具，在保障商家隐私和数据安全的前提下提



供竞争分析。需要强调的是，我们不作无原则的数据披露，在这个产品中，只要涉及其他商家核心数据，均作指数化处理（这一方法同样被应用在整个生意参谋平台中），而非赤裸裸地呈现；在分析竞争群体时，则以群体均值的形式进行呈现，且“群体”一般是10个店铺以上，不是少量几个店铺，以此解决商家了解竞争环境的需求，也避免店铺核心数据被“窥探”。

从我情，到行情，再到敌情，这三个层次、三个维度的数据披露和分析，在一定程度上可以满足大多数商家对店铺经营数据的基本诉求。其实数据服务并不是提供得越多越好，还要注重数据的统一性、及时性和准确性；否则，数据提供越多，给商家带来的困扰可能就越大。

在这点上，平台背后看不见的“数据中台”给生意参谋提供了大量技术保障。例如，在体验方面，生意参谋的数据来自阿里巴巴大数据公共层 OneData。OneData 可以对集团内外数量繁多的数据进行规范化和数据建模，从根本上避免数据指标定义不一致、重复建设的问题，从而确保生意参谋对外数据口径标准统一，计算全面、精确。商家不用再纠结从多个数据产品看到的数据不一致，也不会再遇到小二看到的数据和自己看到的不一样的情况。

在技术层面，数据中台的实时数据计算技术也可以保障生意参谋众多数据指标的实时性和准确性。商家这一秒就能看到上一秒的数据，一旦店铺出现异常，可以马上发现并进行处理。

在用户洞察方面，基于阿里大数据团队全力打造的 OneID 体系，商家可以在生意参谋上更好地洞察消费者画像。

以真人识别为例，顾客 A 白天在电脑上打开了某旗舰店的店铺页，浏览并看中店内某款商品，加了购物车；晚上躺在床上用手机打开淘宝，把购物车中的商品支付了。在这个案例中，顾客 A 分别使用电脑和手机浏览了某旗舰店，这在很多数据工具中，顾客 A 可能会被识别成两个人，而生意参谋会将这个访客的行为轨迹（包括跨屏的行为）串联起来，识别出独立用户个体，而非简单的用户行为次数。

阿里数据中台顶层设计如图 16.3 所示。OneData 体系、OneID 技术等在其中为生意参谋等数据产品提供了稳定的技术支持。

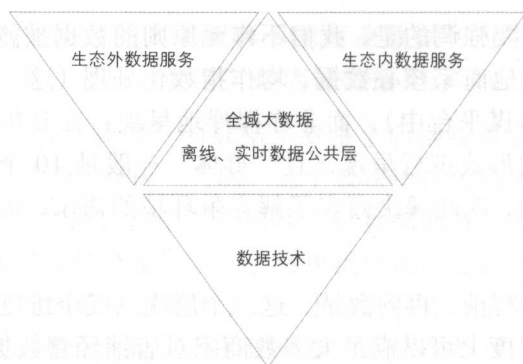


图 16.3 阿里数据中台顶层设计图

诸如以上技术，在生意参谋上的体现还有很多，此处就不一一赘述了。可以说，作为阿里巴巴大数据在商家端的重要体现，生意参谋不仅是商家统一的数据产品平台，更是一系列数据产品技术能力组件、业务分析方法论、运营服务体系的整体构成。

### 16.1.3 商家应用实践

一方面，生意参谋不断开放阿里巴巴的大数据能力；另一方面，其服务对象正受益于大数据，基于数据不断提升自身运营效率和获客能力。

比如周黑鸭食品旗舰店经营的商品以短保商品为主，这类商品的特点是保质期短，对储存温度、包裹温度都有特殊要求。如果备货太多，商品容易过期；如果备货过少，又影响发货速度，进而影响客户体验。

在使用生意参谋之前，周黑鸭备货主要靠经验，几百盒甚至上千盒产品报废的情况经常发生。但通过生意参谋，周黑鸭可以实时监控店铺商品被加入购物车的数量明细，从而精准预测销量。2015年“双11”期间，周黑鸭通过平台预测的大促销售额是2000万，与实际销售额2150万高度吻合。大促7天之内，2000多万商品全部发货完毕，没有出现任何因定量不精准而导致的产品过期问题。

烟花烫是天猫平台上的原创设计女装品牌，每季度会推出新品大约300款，但因为资源所限，新款开发完毕后，还需要对其进行再区分，

进而确定优、良、中、差等级来搭配不同的运营策略。

鉴于此,烟花烫基于生意参谋和相关数据,独创了一套“赛马”机制。首先,新品上线后,运营团队会通过生意参谋了解不同宝贝的访客、成交、加购、收藏和页面停留情况,然后将对应指标数据标准化,再进行加权计算,从而形成初步的“成绩单”。对“成绩单”内访客数高且综合分高的商品,运营团队逐步提高访客数;对访客数高但综合分低的商品,则逐步降低访客数,同时考虑更大优惠/搭配调整,提高转化率。“赛马”机制启动后,通过半年的调整优化,烟花烫天猫旗舰店的销售额同比增长了100%。

需要注意的是,当前商家对数据的应用已经不局限在运营层面,还延伸到团队激励、品牌传播等层面。近两年“双11”期间,三只松鼠、太平鸟、马克华菲等商家还会基于生意参谋数据作战室搭建舞台,并邀请媒体、合作伙伴等前来观战,和员工一同感受“双11”期间数字跳动的气氛,进而展示自身品牌实力。

类似的案例其实不少,从这些商家的应用实践来看,数据对商家的作用是毋庸置疑的。随着数据逐渐成为阿里巴巴赋能商家的新能源,掌握数据并基于数据驱动运营,是商家保持市场竞争力的关键。未来,只有把数据同商业实践、消费者行为结合起来,并利用其洞察消费者画像,挖掘市场潜在需求,才有可能触发更高层次的消费场景。

未来,生意参谋在全渠道、全链路、个性化和智能化等方面还会不断探索,包括打通阿里巴巴内多店数据、阿里巴巴外电商数据、线下商业数据,做好全渠道数据的采集计算;除了现有的经营数据外,还将提供财务分析、会员画像等更多环节的数据服务,进一步丰富全链路服务监测;同时,还会尝试打造更个性化的数据分析门户,建立可视化组件库,建立智能化业务数据预警监测体系等。

而在电商外领域,也会加速布局,力求阿里巴巴的业务走到哪里,数据服务就跟到哪里。我们希望,未来生意参谋能基于数据打通,建立各个业态商业群体之间的连接,使得商业价值最大化。同时,面向不同业态、不同群体,提供持续、分层次、分场景、快速、低成本的数据赋能。

阿里数据中台视角的生意参谋全景如图16.4所示。

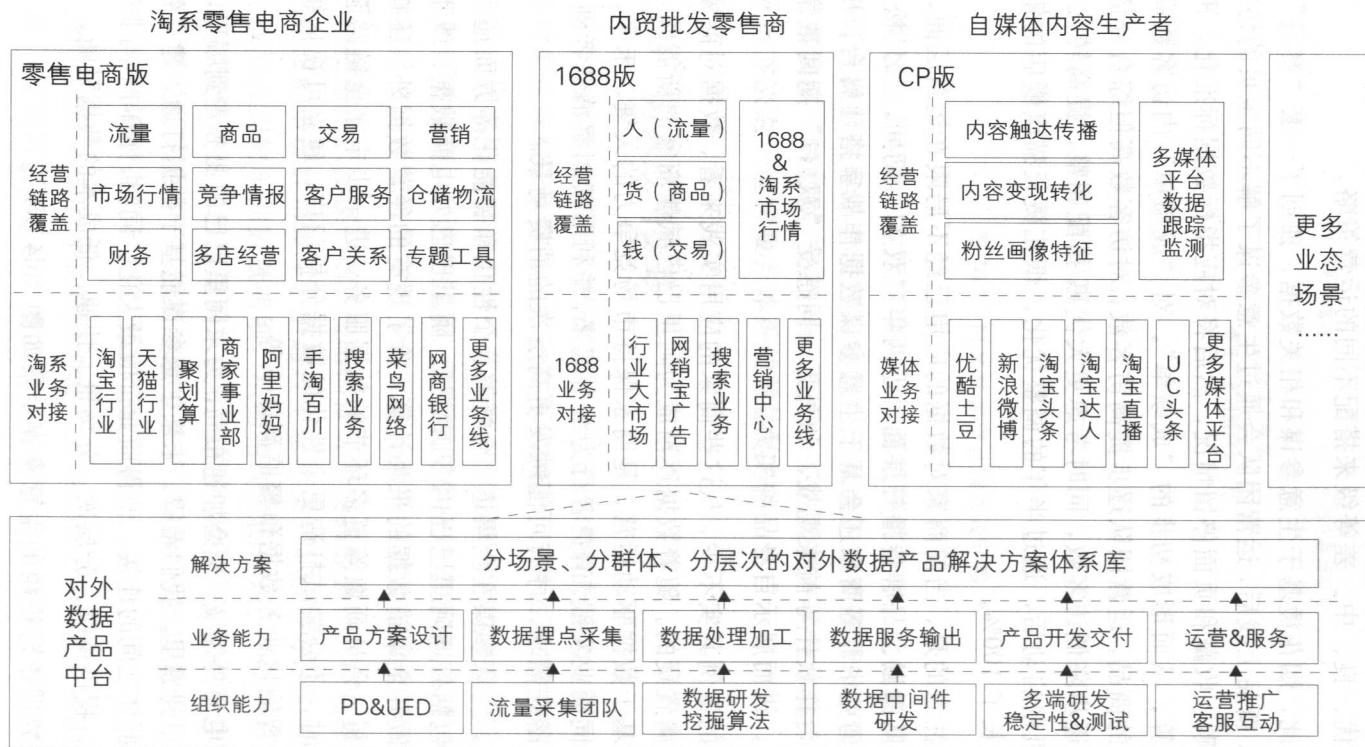


图 16.4 阿里数据中台视角的生意参谋全景图

## 16.2 对内数据产品平台

在阿里巴巴内部，数据分析基本上是所有员工必备的技能。本节主要介绍阿里巴巴对内数据产品的由来、整体架构，以及它是如何有效支持内部员工的日常工作的。

### 16.2.1 定位

在 DT 时代，数据作为商业的“水、电、煤”，需要通过各种各样的“管道”将“水、电、煤”输出给商业、赋能商业，其中数据产品就是一类非常重要的“通道”，通过数据产品将数据转化为用户更优做决策和行动的依据。数据产品有多种形态，包括最简单常见的报表（如静态报表、Dashboard 等简单统计分析）、多维分析（OLAP、即席查询分析等工具型数据产品）、专题分析型数据产品（面向某一类业务场景，沉淀分析思路）、智能型数据应用产品（如个性化搜索、推荐、广告定向推送、精准营销等，这类数据应用产品的发展较为成熟，且大都在数据外面穿了一层外衣，使非专业的用户并不能直观地感受到它是数据产品）。本节会重点介绍阿里巴巴赋能企业内部高管和小二的自助报表产品和商业分析产品。

数据产品的本质是产品，既然是产品，那么首先要回答用户是谁，用户的痛点是什么，产品要解决用户的哪些痛点，即产品给用户带来的价值是什么。对于企业内部数据产品，它的用户是公司的员工，包括销售、BD、运营、产品、技术、客服、管理者等多种角色；解决的痛点是用户对业务发展中的数据监控、问题分析、机会洞察、决策支持等诉求，提供给用户高效率获取数据、合理分析框架、数据辅助业务决策的价值。在阿里巴巴内部，针对不同业务、不同层级的用户、不同的使用场景，会有不同的数据需求，需要对不同业务、不同使用场景的数据需求进行高度抽象，同时又要深入业务场景，规划设计出既要拓展性强，又要贴合业务，还要开发效率高的数据产品。

## 16.2.2 产品建设历程

阿里巴巴对数据产品建设不是一蹴而就的，而是伴随着业务、大数据行业的高速发展一起成长起来的，整个对内数据产品建设大概经历了四个阶段。

### 1. 临时需求阶段

时间回溯到 2003 年左右，当时大数据的概念还未兴起，数据化运营还未被提及，用户对数据的诉求也很简单，即获取业务现状的基本数据。当时还没有数据产品的概念，用户诉求主要以临时取数方式满足，用户需要了解业务基本数据时，提交需求给数据仓库团队，数据工程师通过编写代码，将数据跑好，再给到用户，基本靠人力在做支持。这个阶段逐渐积累了数据和业务经验，为后续数据产品的思考和规划积累了丰富的原始需求。

### 2. V2.0 自动化报表阶段

到了 2006 年，随着业务的不断发展壮大，业务对数据的需求越来越强烈，数据工程师“人肉”支持效率太低，已经跟不上业务和时代的发展。基于之前在数据仓库技术、业务支撑方面的经验积累，数据仓库团队将相似的需求合并同类项的同时总结提炼，并且引入 BI 工具，通过报表和 Dashboard 的方式将数据需求固化下来，进而实现了自动化。一次开发可以满足更多的用户，除了表格之外，还有简单的可视化，不但减少了数据仓库团队的重复建设，提高了用户获取数据的效率，还增强了数据的可读性，让更多人学会使用和分析数据。

当然，严格来说，这个阶段还不算有真正的数据产品，从当时的成本、效益方面考虑，阿里巴巴数据仓库选择购买微策略的 MSTR 工具，后来又升级为 BIEE 工具，用于报表和集市的制作。为了方便用户找到报表，搭建了数据门户，将报表和集市以业务主题的方式进行组织，数据门户雏形在这个阶段开始出现；发展到后面，用户不断加深对数据价值的认知，需求越来越精细化和多元化，对数据化运营的期待越来越高，

单纯的报表监控分析已经无法高效满足当时的用户诉求，而且受限于 BIEE 工具，拓展性和报表体验难以提升……但是在这个阶段，建设团队积累了更丰富的技术和业务经验。基于此，阿里巴巴数据人已经开始思考数据产品化之路。

### 3. 自主研发 BI 工具阶段

从 2012 年年底开始，我们有了真正意义上的数据产品。随着业务的高速发展，数据维度和数据量增长迅猛，数据需求越来越繁杂和参差不齐，同时不能靠增加人力来满足，所以急需强大、易用的 BI 工具来支持。

受限于第三方 BI 工具，MSTR、BIEE 主要面向的是专业数据人员，而我们需要能有工具可以面向普通的数据使用人员，满足其自身不同的需求；当时阿里巴巴已经将大部分数据迁移上云，经典的第三方 BI 工具无法直接接入云上的数据。基于上述两个考虑，我们最早构思了一个“取数机器人”的工具，目的就是想让这个“机器人”帮大家建数据和获取数据，降低大家对数据需求的成本。围绕这个构思，我们规划落地了“快门”和“小站”（“快门”和“小站”均为内部数据产品名称，仅限于在阿里巴巴内部使用）。

“快门”的寓意是获取数据可以像相机的快门一样快速、便捷；“小站”的寓意是方便用户搭建出自己或部门的数据站点。“快门”和“小站”的定位是阿里巴巴内部的 BI 工具。快门端同时支持用户标准模式和 SQL 模式，其中标准模式是拖拉拽的方式，主要针对开发好的集市宽表，支持“亿级数据，秒级响应”，用户只需要拖拉拽，再配置一下简单的 COUNT 或 SUM 就可以获取到自己的数据；SQL 模式更灵活，将 SQL 代码代入进去，即可生成一张报表，同第二阶段的自动化报表功能类似。小站则是把快门的报表和外部的数据或页面，以整体的分析思路、可视化（表格、图表等）和解读（富文本）的方式更好地组织和展示出来。

这个阶段，是真正意义上的自主研发数据产品或工具的阶段，是不断积累业务需求及产品规划和技术的过程，为后续做数据平台打下了坚实的基础。



#### 4. V4.0 数据产品平台

2014 年前后，“快门”和“小站”经过两年时间的打磨，在产品功能、体验方面都有很好的提升，不仅为淘宝业务提供服务，同时还为整个阿里巴巴电商业务提供服务，在整个阿里巴巴集团都有一定的影响力。

随着业务的发展，很多垂直业务的数据团队合并，规避重复建设，更好地形成合力，提升效能；同时业务对数据的诉求越来越精细化和多元化，除了使用 BI 工具方便获取数据，做数据日常监控之外，还需要将特定业务场景的分析思路固化下来，沉淀在产品里，以自动化来替换分析师的“人肉”分析，同时需要把业务数据应用到线上。比如众所周知的个性化搜索和推荐、精准营销、选品选商、搭建专场、数据效果跟踪等一整套闭环分析需求……所以对平台化的思考越来越清晰，“阿里数据平台”应运而生，目的是为阿里巴巴内部用户打造一站式数据获取、数据分析、数据应用的数据产品平台。同时随着无线发展的大趋势，随时随地掌握数据的需求也越来越明显，所以阿里数据平台在一开始就布局了 PC 版和 APP 版的规划。

这一年开始，数据产品可谓百花齐放。例如，“双 11”大促，需要有实时数据监控分析，不仅仅是对大盘的监控，还要有对各个会场、商家、商品的实时流量和成交的监控，以便调整资源和调配流量分配。对于这类需求，传统的数据仓库离线计算是满足不了的。为了更好地解决这类需求，我们突破传统数据仓库的边界，研究业界前沿的流计算技术，同时规划设计直播类产品，不仅覆盖了实时交易，还有实时流量，从大盘到行业再到会场、商家、商品各个维度；同时产品从淘宝业务逐步拓宽到阿里巴巴电商的所有业务，实时指导大促各个层级用户的数据化运营。

再如，互联网公司对网站流量分析是运营最基本的诉求，运营需要了解当日访问页面的流量、何时访问量最高、每个位置的点击效果与引流效果等。针对流量分析，我们规划设计了类似于 GA 的流量分析产品，但定位跟 GA 不同，其不只是流量分析的通用分析产品，覆盖了 GA 的流量基础统计分析的功能，同时还是阿里巴巴电商流量引导转化深度分析的产品，将流量数据同交易引导转化数据打通，让运营更有抓手。

在这个阶段，我们根据不同的业务场景，规划落地了较多场景化或



专题类的分析型数据产品，这类产品的适用场景和用户非常聚焦，除了解决数据获取效率之外，还提供了特定业务场景的分析思路，使数据价值更加突显。

针对众所周知的个性化搜索和推荐、精准营销、选品选商、搭建专场、数据效果跟踪等一整套闭环分析需求，我们规划了对内使用的分析产品，不仅能够通过勾选条件筛选出目标数据，同时还能够做自助分析，调整条件满足需求后直接对接前台应用系统，实现个性化、精准营销、选品选商、搭建专场的需求。

这个阶段还在继续探索中，日渐丰富和复杂的业务场景也对产品提出了更高的要求，产品之间如何打通，是当前我们在设计和规划数据产品中着重思考和突破的地方。

### 16.2.3 整体架构介绍

在16.2.2节中，介绍了阿里巴巴对内数据产品的四个发展阶段，本节会详细介绍第四阶段（也是当前阶段）的平台和产品架构，以让读者对阿里巴巴对内数据产品有更深入的认识。

对于数据产品，数据一直都是产品的核心和生命力，数据质量和数据安全是数据产品最基础的要求，如果一款数据产品提供的数据不准，那么对使用产品的用户来说是灾难性的，甚至会根据数据产品得出完全错误的结论，进而影响业务决策。

数据安全的管理在阿里巴巴内部是非常严格的，所以数据权限管控的重要性不言而喻。阿里巴巴对内数据产品基于阿里数据公共层，数据公共层在数据准确性方面投入了很多精力，同时有一整套指标算法统一和研发工具来保障，取得了非常不错的成果。这部分内容在数据模型建设、数据加工处理环节已经详细阐述，这里也不再赘述。

阿里巴巴对内数据产品平台，即阿里数据平台，包括PC版和APP版，共有四个层次，即数据监控、专题分析、应用分析和数据决策，如图16.5所示。

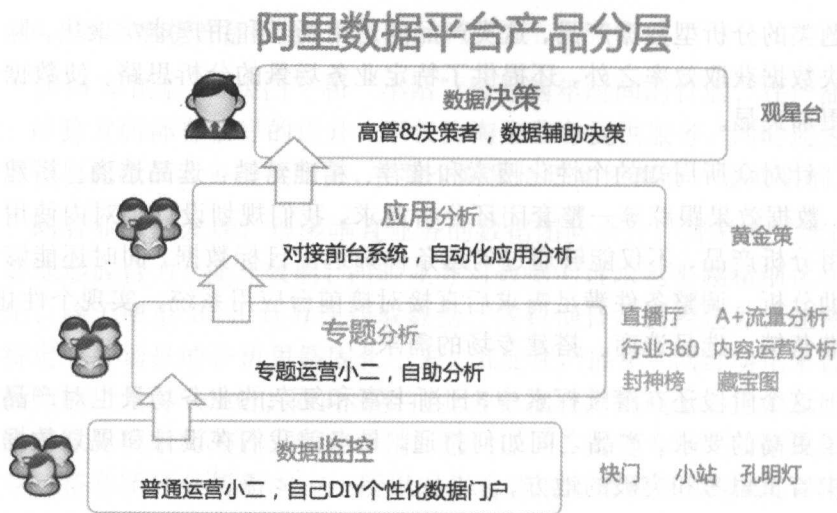


图 16.5 阿里数据平台整体架构图

## 1. 数据监控

对于所有内部普通运营小二, 都有查看或分析业务数据的需求, 阿里数据平台提供最基础的报表工具, 供用户自助取数、多维分析、DIY 个性化数据门户。这块对应的产品主要是我们自主研发的 BI 工具, 同时我们还在研发一款基于 Web Excel 的 BI 工具——孔明灯, 除了服务内部之外, 还可以将内部的 BI 工具能力对外输出给阿里巴巴的商家和合作伙伴使用。

## 2. 专题分析

对于专题运营小二, 如行业运营小二, 对类目有强烈的分析诉求, 按照分析师沉淀的成熟分析思路组织数据, 实现行业运营小二自助分析行业异动原因, 发现行业潜在机会, 实现“人人都是分析师”, 提高数据化运营的效率和质量。这块对应的产品主要有实时直播分析产品, 可以根据实时数据调整资源及流量分配; 行业一体化分析产品, 从行业视角提供行业 360 度的数据, 同时沉淀行业分析思路; 流量分析产品, 从流量的角度提供流量相关数据, 包括对站点、页面、区块、位置的浏览,

曝光、点击分布，以及获得资源位的活动投放等数据。

### 3. 应用分析

对于很多业务系统的流程，数据是其中不可缺少的一环，通过对接前台系统，实现数据的自动化。比如日常营销活动运营，需要选品选商、搭建专场，那么如何选商选品，以及选择什么样的商家和商品，对整个活动非常重要，完全人工筛选在效率上会受到很大制约。为解决此问题，我们提供了专门产品来完成系统对接，不只是数据的对接，同时也包含产品交互间的打通。这一产品不仅能够通过勾选条件筛选出目标数据，同时还能够做自助分析，调整条件满足需求后直接对接前台应用系统，实现个性化、精准营销、选品选商、搭建专场的需求。

### 4. 数据决策

对于高管和决策者，既需要宏观的业务数据，又需要可下沉的数据，还需要丰富的趋势数据来辅助决策，需要通过数据了解业务进展、当前进展是否合理、接下来的业务方向等，针对此类需求提供定制化的数据产品供决策参考，为高管提供宏观决策分析支撑平台，分析历史数据规律，预测未来发展趋势，洞察全行业动态。

随着阿里巴巴业务的发展、新技术的引入，对内数据产品势必会不断迭代，去探索更多、更新的数据价值，更高效地开发数据产品。未来，对内数据产品平台的发展会重点在两个方面进行突破：一方面，把 BI 工具等工具型产品功能做强做大，不仅可以做报表，还可以做出专题分析型产品，在有限的研发资源下，更高效地实现数据产品；另一方面，在应用型数据产品上做更多的探索，赋能业务数据化的运营。

# 附录 A

## 本书插图索引

第 1 章 总述.....1	第 4 章 离线数据开发.....48
图 1.1 阿里巴巴大数据系统体系架构图.....3	图 4.1 MaxCompute 体系架构图.....50
第 1 篇 数据技术篇	图 4.2 开发工作流图.....53
第 2 章 日志采集.....8	图 4.3 对应于开发工作流的产品和工具.....54
图 2.1 一次典型的互联网页面请求- 响应过程.....10	图 4.4 SQLSCAN 工作流程图.....55
图 2.2 阿里巴巴页面浏览日志采集 方案流程框架.....13	图 4.5 DQC 工作流程图.....56
图 2.3 Native 和 H5 日志桥接示意图.....21	图 4.6 使用在彼岸进行回归测试流程图.....58
图 2.4 数据处理全链路.....27	图 4.7 调度任务关系依赖图.....59
第 3 章 数据同步.....29	图 4.8 数据开发流程与调度系统 的关系图.....60
图 3.1 直连同步示意图.....30	图 4.9 任务状态机模型关系图.....61
图 3.2 数据文件同步示意图.....31	图 4.10 工作流状态机模型关系图.....62
图 3.3 数据库日志解析同步示意图.....32	图 4.11 调度引擎工作原理示意图.....63
图 3.4 DataX 可接入的数据源.....36	图 4.12 执行引擎逻辑架构图.....63
图 3.5 DataX 架构设计图.....37	第 5 章 实时技术.....68
图 3.6 TimeTunnel 实时数据传输示意图.....38	图 5.1 数据直播大屏.....69
图 3.7 分库分表处理.....40	图 5.2 流式技术架构图.....72
图 3.8 TDDL 分布式数据库访问引擎.....40	图 5.3 消息系统与数据中间件、 数据库之间的关系.....74
图 3.9 淘宝交易订单同步示意图.....43	图 5.4 实时应用的整个拓扑结构图.....75
	图 5.5 数据流向.....82

图 5.6 多流关联示例 .....	83	图 9.5 原子指标详情 .....	143
图 5.7 强保障数据多链路冗余示意图 .....	87	图 9.6 派生指标命名示例 .....	144
图 5.8 多机房容灾示意图 .....	89	图 9.7 派生指标示例 1 .....	147
<b>第 6 章 数据服务 .....</b>	<b>91</b>	图 9.8 派生指标示例 2 .....	147
图 6.1 阿里数据服务架构演进过程 .....	92	图 9.9 模型层次关系图 .....	149
图 6.2 DWSOA 架构示意图 .....	92	图 9.10 模型架构图 .....	150
图 6.3 OpenAPI 架构示意图 .....	93	图 9.11 实施 workflow .....	155
图 6.4 SmartDQ 架构示意图 .....	94	<b>第 10 章 维度设计 .....</b>	<b>159</b>
图 6.5 统一的数据服务层 (OneService) 架构示意图 .....	96	图 10.1 规范化处理淘宝商品维度 所表现的形式 .....	164
图 6.6 SmartDQ 的元数据模型架构 示意图 .....	97	图 10.2 反规范化处理淘宝商品维度 所表现的形式 .....	165
图 6.7 SmartDQ 架构示意图 .....	98	图 10.3 淘宝交易事实表维度 .....	179
图 6.8 iPush 应用架构示意图 .....	100	图 10.4 类目维度 .....	181
图 6.9 Lego 应用架构示意图 .....	101	图 10.5 类目 .....	183
图 6.10 uTiming 应用架构示意图 .....	102	图 10.6 类目树 .....	183
图 6.11 场景一 .....	104	图 10.7 采用多字段方式处理多值维度 .....	186
图 6.12 场景二 .....	104	图 10.8 采用桥接表方式多理多值维度 .....	187
图 6.13 查询拆分 .....	105	图 10.9 淘宝商品 SKU 和属性信息示例 .....	187
图 6.14 主处理模块处理步骤 .....	107	图 10.10 卖家主营类目维度设计 .....	188
图 6.15 使用缓存流程图 .....	108	图 10.11 杂项维度 .....	189
图 6.16 REPLACE 语法 .....	109	<b>第 11 章 事实表设计 .....</b>	<b>190</b>
图 6.17 不同应用环境下的三套元数据 .....	112	图 11.1 淘宝交易订单的流转过程 .....	194
<b>第 7 章 数据挖掘 .....</b>	<b>116</b>	图 11.2 父子订单合并成一条记录 .....	198
图 7.1 MaxCompute MPI 处理流程图 .....	118	图 11.3 父子订单拆分成多条记录 .....	198
图 7.2 阿里巴巴数据挖掘中台 .....	121	图 11.4 淘宝交易事务事实表 (确定维度) .....	199
图 7.3 常见的数据挖掘应用 .....	123	图 11.5 冗余维度的淘宝交易 事务事实表 .....	200
<b>第 2 篇 数据模型篇</b>		图 11.6 1688 交易订单下单事务事实表 .....	201
<b>第 8 章 大数据领域建模综述 .....</b>	<b>130</b>	图 11.7 1688 交易订单支付事务事实表 .....	201
图 8.1 Data Vault 模型实例 .....	135	图 11.8 1688 交易订单详情实例 .....	202
图 8.2 Anchor 模型图 .....	136	图 11.9 1688 交易订单下单事务 事实表数据实例 .....	202
<b>第 9 章 阿里巴巴数据整合及管理体系 .....</b>	<b>138</b>	图 11.10 1688 交易订单支付事务 事实表数据实例 .....	202
图 9.1 体系架构图 .....	139	图 11.11 淘宝交易多事务事实表 .....	204
图 9.2 规范定义实例 .....	140		
图 9.3 派生指标 .....	142		
图 9.4 业务过程命名示例 .....	143		

图 11.12	淘宝交易订单详情实例	204	图 13.3	TPC-H 测试对比图	255
图 11.13	淘宝交易多事务事实表 数据实例	204	图 13.4	SQL/MR 在 MaxCompute 中的 细分步骤	256
图 11.14	收藏商品明细	206	图 13.5	Map 端的两个主要过程示意图	257
图 11.15	收藏商品事务事实表数据实例	206	图 13.6	LogView 日志 (一)	259
图 11.16	淘宝交易卖家快照事实表	211	图 13.7	LogView 日志 (二)	261
图 11.17	淘宝卖家历史至今快照事实表	213	图 13.8	Map 和 Reduce 阶段的执行过程	262
图 11.18	淘宝商品历史至今快照事实表	214	图 13.9	Fuxi Task 的详细执行信息	263
图 11.19	淘宝买卖家历史至今快照 事实表	214	图 13.10	Instance 读入的数据量	263
图 11.20	卖家好中差评以及星级	215	图 13.11	在多种时间范围内分终端 统计买家数和商品数代码	271
图 11.21	卖家 DSR 评分	215	图 13.12	LogView 日志	271
图 11.22	淘宝卖家信用分和 DSR 快照事实表	216	图 13.13	整体运行的 LogView 日志	273
图 11.23	淘宝好中差评快照事实表	217	第 14 章	存储和成本管理	275
图 11.24	淘宝交易累积快照事实表 (确定事实前)	220	图 14.1	数据重分布方法	277
图 11.25	淘宝交易累积快照事实表 (确定事实后)	221	图 14.2	存储治理项优化主要流程图	278
图 11.26	淘宝交易事务事实表	231	图 14.3	MaxCompute 数据表依赖 关系示意图	282
图 11.27	按商品聚集的星形模型	232	图 14.4	数据刷新链路示意图	283
图 11.28	按卖家聚集的星形模型	232	第 15 章	数据质量	285
图 11.29	按卖家、买家、商品聚集 的星形模型	233	图 15.1	数据质量保障原则	286
图 11.30	按二级类目聚集的星形模型	234	图 15.2	数据质量建设方法	288
图 15.3	数据流转过程	291	图 15.4	生意参谋产品打标过程	292
图 15.4	生意参谋产品打标过程	292	第 4 篇	数据应用篇	
第 3 篇	数据管理篇		第 16 章	数据应用	304
第 12 章	元数据	236	图 16.1	生意参谋产品理念图	306
图 12.1	统一元数据体系建设思路图	238	图 16.2	生意参谋功能架构图	307
图 12.2	Data Profile	240	图 16.3	阿里数据中台顶层设计图	310
图 12.3	驱动 ETL 开发示意图	243	图 16.4	阿里数据中台视角的生意 参谋全景图	312
第 13 章	计算管理	245	图 16.5	阿里数据平台整体架构图	318
图 13.1	优化器各模块协调工作图	250			
图 13.2	Volcano Planner 工作原理图	253			



## 好书分享



《尽在双11：阿里巴巴技术演进与超越》

ISBN 978-7-121-30917-5

阿里巴巴集团双11技术团队 著

精炼揭秘双11八年技术演进史

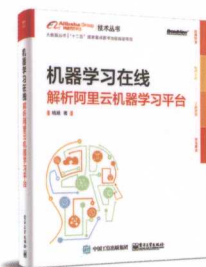


《技术之瞳：阿里巴巴技术笔试心得》

ISBN 978-7-121-29933-9

阿里巴巴集团校园招聘笔试项目组 著

进击阿里巴巴技术岗的精准指南

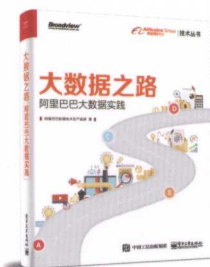


《机器学习在线：解析阿里云机器学习平台》

ISBN 978-7-121-31869-6

杨旭 著

永不掉线的机器学习实践库，助你快速掌握AI技术！



《大数据之路：阿里巴巴大数据实践》

ISBN 978-7-121-31438-4

阿里巴巴数据技术及产品部 著

阿里巴巴分享对大数据的认知、与世界共创数据智能的重要基石。



本书策划编辑：张彦红（出版喜洋洋）

邮箱：zhanghong@phei.com.cn

欢迎准作者咨询写书、出版事宜

欢迎读者反馈、交流

# 大数据之路

## 阿里巴巴大数据实践

- ◆ 十余年大数据实践总结 ◆
- ◆ 多位阿里数据人经验汇总 ◆

### 数据技术篇

日志采集  
离线数据开发  
数据服务

数据同步  
实时技术  
数据挖掘

### 数据模型篇

大数据领域建模综述  
维度设计

阿里巴巴数据整合及管理体系  
事实表设计

### 数据管理篇

元数据  
存储和成本管理

计算管理  
数据质量

### 数据应用篇

生意参谋

对内数据产品平台

**易读易查** 知识点目录 + 插图索引

**加深理解** 精心绘制139张流程与架构示意插图

上架建议：大数据



博文视点Broadview



@博文视点Broadview



策划编辑：张彦红  
责任编辑：葛娜  
封面设计：李玲

ISBN 978-7-121-31438-4



9 787121 314384 >

定价：79.00元